

# Chapter 14. Defining Classes

## In This Chapter

In this chapter we will learn how to **define custom classes** and their elements. How to declare **fields**, **constructors** and **properties** for the classes. We will revise what a method is and we will broaden our knowledge about **access modifiers** and **methods**. We will examine the characteristics of the constructors, find out how the program objects coexist in the dynamic memory and how their fields are initialized. Finally, we will explain what the **static elements** of a class are – the fields (including **constants**), properties and methods and how to use them properly. In this chapter we will also introduce generic types (**generics**), enumerated types (**enumerations**) and **nested classes**.

## Custom Classes

A computer program aims at **solving a problem** by implementation of an idea. In order to create a solution, first we sketch a simplified model, which does not represent everything, but focuses on facts, which are significant for the end result. Afterwards, based on the sketched model, we look for an answer (i.e. create an algorithm) to our problem and describe the solution by means of a programming language.

Nowadays, the most used programming languages are object-oriented. **Object-oriented programming (OOP)** is close to the way humans think and readily allows for description of models of the real world. OOP offers tools for modelling concepts, using classes of objects. Different from the .NET system framework, definition of custom classes is a built-in feature of the C# programming language. The purpose of this chapter is to get familiar with it.

## Let's Recall: What Do Class and Object mean?

**Class** in OOP is 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 objects (the copies), which are created from the class (pattern).

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

For example, if we have a class **Dog**, which describes some of the characteristics of a real dog, then the objects based on the description of the class (e.g. the dogs "Fido" and "Rex") are of type **Dog**. It means the same as when the object string "some string" is of the type of the **String** class. The difference is that **objects** of type **Dog** are copies from a custom class, which is not part of the system library classes of the .NET Framework, but defined by ourselves (the users of the programming language).

## What Does a Class Contain?

Every class contains a definition of the kind of data types and objects describing it. The object (a copy of the class) holds the actual **data**. The data defines the object's **state**.

In addition to the **state**, the class describes the **behavior** of the objects. The behavior is represented by actions, which can be performed by the objects themselves. The resource in OOP for describing the behavior of the objects from a class, are the **methods** in the class body.

## Elements of the Class

We will first go through the main elements of every class and later explain them in detail. The main **elements of C# classes** are:

- **Class declaration** – this is the line where we declare the name of the class, e.g.:

```
public class Dog
```

- **Class body** – similar to the method, the classes also have a single body. It is defined right after the class declaration and enclosed in curly brackets "{" and "}". The elements of the class, which are presented below, are the content of the body.

```
public class Dog
{
    // ... The content of the body of the class comes here ...
}
```

- **Constructor** – it is used for **creating new objects**. Here is a typical constructor:

```
public Dog()
{
    // ... Some code ...
}
```

- **Fields** – they are variables declared inside the class (also known as **member-variables**). The data of the object, which these variables represent and retain, is the specific state of an object, required for the proper working of the object's methods. The values of the fields are specific for the object, unless the fields are **static** and values are shared among all objects.

```
// Field definition
private string name;
```

- **Properties** – they describe the **characteristics** of a class. Usually, the values of the characteristics are kept in the fields of the object. Similar to the fields, the properties may be specific to an object or shared among objects.

```
// Property definition
private string Name { get; set; }
```

- **Methods** – from the chapter "[Methods](#)" we know that methods are named blocks of programming code. They perform particular actions and through them the objects achieve their particular behavior, depending on the class type. Methods execute the implemented programming logic (algorithms) and handle the data.

## Sample Class: Dog

Here is how a class looks like. The class **Dog** owns all the elements described so far:

```
// Class declaration
public class Dog
{    // Opening bracket of the class body

    // Field declaration
    private string name;

    // Constructor declaration ( parameterless empty
    constructor)
    public Dog()
    {
    }

    // Another constructor declaration
    public Dog(string name)
    {
        this.name = name;
    }

    // Property declaration
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    // Method declaration (non-static)
    public void Bark()
    {
        Console.WriteLine("{0} said: Wow-wow!",
            name ?? "[unnamed dog]");
    }
}    // Closing bracket of the class body
```

Later in the chapter we will explain the code in greater detail.

## Usage of Class and Objects

In the chapter "[Creating and Using Objects](#)" we saw in detail how new objects of a class are created and how they can be used. We will briefly revise this programming technique.

### How to Use a Class Defined by Us (Custom Class)?

In order to use a class, first we need to create an object from it. This is done by the reserved word **new** in combination with some of the constructors of the class.

If we want to manipulate the newly created object, we will have to assign it to a variable of the same type as the class. By doing so, the variable will keep a connection (reference) to the object.

Using the variable and the "dot" notation, we can call the methods and the properties of the object, as well as gain access to the fields (member-variables).

## Example – A Dog Meeting

Let's use the example from the [previous section](#) where we defined the class **Dog**, describing a dog and add a method **Main()**. In this method we demonstrate how to use mentioned elements. We create some **Dog** objects, assign properties to them and call methods on these objects:

```
static void Main()
{
    string firstDogName = null;
    Console.Write("Enter first dog name: ");
    firstDogName = Console.ReadLine();

    // Using a constructor to create a dog with specified name
    Dog firstDog = new Dog(firstDogName);

    // Using a constructor to create a dog with a default name
    Dog secondDog = new Dog();

    Console.Write("Enter second dog name: ");
    string secondDogName = Console.ReadLine();

    // Using property to set the name of the dog
    secondDog.Name = secondDogName;

    // Creating a dog with a default name
    Dog thirdDog = new Dog();

    Dog[] dogs = new Dog[] { firstDog, secondDog, thirdDog };

    foreach (Dog dog in dogs)
    {
        dog.Bark();
    }
}
```

The output from execution of the code will be the following:

```
Enter first dog name: AxL
Enter second dog name: Bobby
AxL said: Wow-wow!
Bobby said: Wow-wow!
[unnamed dog] said: Wow-wow!
```

In this example program, with the help of **Console.ReadLine()**, we get the names of the objects of type **Dog**, which the user enters.

We assign the first entered string to the variable **firstDogName**. Thereafter we use this variable when we create the first object, - **firstDog**, of type **Dog** from the class **Dog**, by assigning it to the parameter of the constructor.

We create the second object **Dog**, without using a string for the name of the dog in the constructor. With the help of **Console.ReadLine()** we get the name of the dog and then assign it to the property **Name**, using a "dot" convention, applied to the variable, which keeps the reference to the second object from type **Dog** - **secondDog.Name**.

When we create the third object from class type **Dog**, we use for the name of the dog its default value which is **null**. Note that in the **Bark()** method dogs without name (**name == null**) are printed as "[unnamed dog]".

Afterward we create an array of type **Dog** and initialize it with the three newly created dog objects.

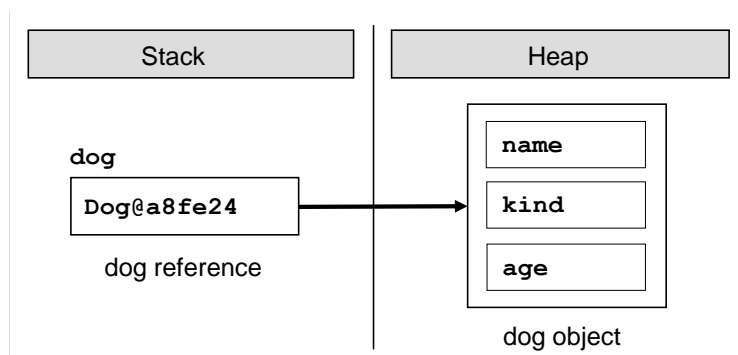
Finally, we use a loop to go through the array of objects of type **Dog**. For every element from the array we again use the "dot" notation when calling the method **Bark()** for each dog object: **dog.Bark()**.

## Nature of Objects

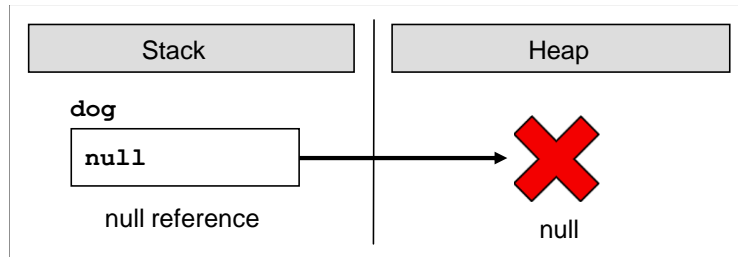
When we create an object in .NET, it consists of two parts. The **significant part (data)** containing the data and is located in the dynamic memory of the operating system (heap) and a **reference part** of this object, which resides in the other part of the operating system's memory, where are stored the local variables and parameters of the methods (the program execution stack).

For example, consider a class called **Dog**, which has the properties name, kind and age. Let's create a variable **dog** from this class. This variable is in the stack memory and is a reference to the dog object that resides in the dynamic memory (heap).

The **reference** is a variable, which can access objects. The figure below depicts an example dog reference, which contains a link to the real **Dog** object in the heap. As compared to the primitive variable (value type), the dog reference variable does not contain the real value (i.e. the data of the object), but the address, where the dog object is located in the heap memory:



When we declare a variable of the type of a class and we do not want the variable to be associated with a specific object, then we assign to it the value **null**. The reserved word **null** in the C# language means that the variable does not point to any object (the value is absent):



## Organizing Classes in Files and Namespaces

In C#, custom classes have to be **saved in files with file extension .cs**. In such a file, several classes, structures and other types can be defined. Although it is not a requirement of the compiler, it is recommended that **every class be stored in exactly one file with the same name**, i.e. the class **Dog** should be saved in a file called **Dog.cs**.

## Organizing Classes in Namespaces

As explained in the chapter "[Creating and Using Objects](#)", the **namespaces in C# are named groups of classes**, which are logically connected, without requirement for specific storage in the file system.

If we want to include in our code namespaces for the operation of our classes, declared in some file or set of files, this should be done by the so called **using directives**. They are not required, but if applied, they should appear on the first lines of the class file, before the declaration of classes and other types. In the next paragraphs we will see how they are exactly used.

There is no requirement to declare classes in a namespace, but it is a good programming technique. The distribution of classes in namespaces allows for better organization of the code and definition of different classes with the same name.

The namespaces contain classes, structures, interfaces and other types of data, as well as other (nested) namespaces. An example of namespace nesting is **System**, which contains the namespace **Data**. The full name of the second namespace is **System.Data** and it is nested in the namespace **System**.

The **full name of a class** in .NET Framework is the class name, preceded by the name of the namespace in which the class is declared, e.g.: **<namespace\_name>.<class\_name>**. By means of the reserved **using** word we can use types from a certain namespace, without having to write the full name, e.g.:

```
using System;  
...  
DateTime date;
```

Instead of:

```
System.DateTime date;
```

A typical declaration sequence, which we should follow when creating custom classes in **.cs** files, is:

```
// Using directives - optional
```

```

using <namespace1>;
using <namespace2>;

// Namespace definition - optional
namespace <namespace_name>
{
    // Class declaration
    class <first_class_name>
    {
        // ... Class body ...
    }

    // Class declaration
    class <second_class_name>
    {
        // ... Class body ...
    }

    // ...

    // Class declaration
    class <n-th_class_name>
    {
        // ... Class body ...
    }
}

```

The declaring of the namespace and the relevant include of it has been explained in the chapter "[Creating and Using Objects](#)" and will not be discussed again.

Note that, the first line of above snippet is a source code comment. Comments are "removed" from the code during compilation and thus the first code line is the using directive.

## Encoding of Files and Using of Cyrillic and Unicode

When creating a **.cs** file, in which to declare our classes, it is good to think about its **character encoding** in the file system.

In the .NET Framework, the compiled code is represented in Unicode so that it is possible to use characters in our code from alphabets other than Latin. In the next example we use Cyrillic letters for identifiers and comments in the code, written in Bulgarian:

```

using System;

public class EncodingTest
{
    // ТЕСТОВ КОМЕНТАР
    static int ГОДИНИ = 4;

    static void Main()
    {

```

```

        Console.WriteLine("years: " + години);
    }
}

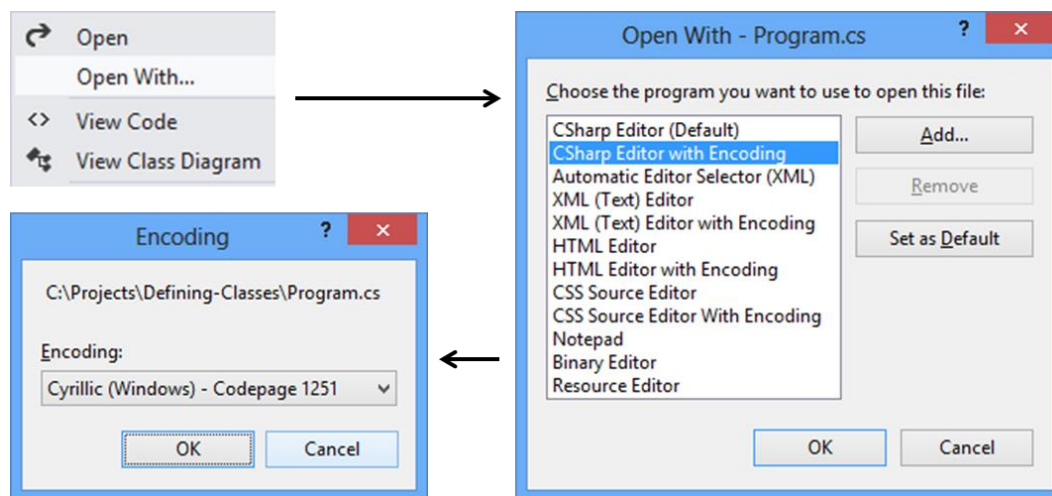
```

This code will compile and execute without a problem. To read the characters by the Visual Studio editor we need to provide an appropriate **encoding of the file**.

As we have seen in the "[Strings](#)" chapter, not all characters can be stored in some encodings. If we use non-standard characters such as Chinese, Cyrillic or Arabic letters, we need to use **UTF-8** or other character encoding that supports these characters. By default, Visual Studio uses the character encoding (system locale) defined in the regional settings of Windows. This is ISO-8859-1 for Latin script, as in the U.K. or the U.S., and Windows-1251 for Cyrillic script, as in Bulgaria.

To use a different encoding other than the system's default encoding in Visual Studio, we need to choose an appropriate encoding for the file before opening it in the editor:

1. From the **File** menu we choose **Open** and then **File**.
2. In the **Open File** window, we click on the option next to the button **Open** and we choose **Open With...**
3. From the list in the **Open With** window, we choose an editor with encoding support, for example **CSharp Editor with Encoding**.
4. Then press **[OK]**.
5. In the window **Encoding**, we choose the appropriate encoding from the dropdown menu **Encoding**.
6. Then press **[OK]**.



The steps for saving files with a specific encoding in the file system are:

1. From the **File** menu we choose **Save As**.
2. In the window **Save File As**, we press the drop-down box next to the button **Save** and choose **Save with Encoding**.
3. In **Advanced Save Options** we select the desired encoding from the list (preferably the universal UTF-8).
4. From the **Line Endings** we select the desired line ending type.



Although we have the ability to use characters from any non-English alphabet in **.cs** files, it is highly recommended to **write all the identifiers and comments in English**, because in this way, our code will be readable by more people worldwide.

Imagine that you live in Germany and you need to read a code written by a Vietnamese person, where the names of all variables and comments are in Vietnamese. You will prefer English, right? Then think about how a developer from Vietnam might have trouble handling variables and comments in German.

## Modifiers and Access Levels (Visibility)

From the chapter "[Methods](#)" we know that a **modifier** is a reserved word and with the help of it we add additional information for the compiler in front of the code related to the modifier.

In C# there are four **access modifiers**: **public**, **private**, **protected** and **internal**. The access modifiers can be used only in front of the following elements of the class: class declaration, fields, properties and methods.

## Modifiers and Access Levels

. With the four access modifiers – **public**, **private**, **protected** and **internal** we control the access to (visibility of) the elements of the class, in front of which they are used. The levels of access in .NET are **public**, **protected**, **internal**, **protected internal** and **private**. In this chapter we will review in detail only **public**, **private** and **internal**. We will learn more about **protected** and **protected internal** in the chapter "[Object-Oriented Programming Principles](#)".

### Access Level "public"

When we use the modifier **public** in front of some element, we are telling the compiler that this element **can be accessed from every class**, even from outside the current project (assembly) or current namespace. The access level **public** defines the absence of restrictions regarding the visibility. This access level is the least restricted access level in C#.

### Access Level "private"

The access level **private** defines **the most restrictive level of visibility** of the class and its elements. The modifier **private** is used to indicate that the element, for which it is issued, **cannot be accessed from any other class** than the class in which it is defined, even if the other class exists in the same assembly. This is the default access level, i.e. it is used when there is no access level modifier placed in front of the respective elements of the class (this is true only for elements belonging to a class).

### Access Level "internal"

The modifier **internal** is used for limiting class element access to files **from the same assembly**, i.e. the same project in Visual Studio. When we create several projects in Visual Studio, their classes will be compiled in different assemblies.

## Assembly

.NET assemblies are **collections of compiled types** (classes and other types) and **resources** forming a logical unit. Assemblies are stored in a binary file of type **.exe** or **.dll**. All types in C# and generally in .NET Framework can reside only inside assemblies. During

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.

## Declaring Classes

The definition of a class is based on strict syntactical rules:

```
[<access_modifier>] class <class_name>
```

When we declare a class, it is mandatory to use the reserved word **class**. After it must follow the name of the class **<class\_name>**.

Besides, the reserved word **class** and the name of the class, several modifiers can be used in the declaration of the class,.

## Class Visibility

Let's consider two classes – **A** and **B**. We say that class **A** can access the elements of class **B**, if the first class can do at least one of the following:

1. Create an object (instance) from class type **B**.
2. Access distinct methods and fields in class **B**, based on the access level assigned to these methods and fields.

Another operation on classes depending on visibility is called **inheritance of a class**. We will discuss this later in the chapter [Object-Oriented Programming Principles](#).

The access level term defines "**visibility**". If class **A** cannot "see" class **B**, the access level of the methods and the fields in **B** does not matter to class **A**.

The access levels, which an outer class can have, are only **public** or **internal**. [Inner classes](#) can also be defined with other access levels.

### Access Level "public"

If we declare a class with access modifier **public**, we can reach it from **every class and from every namespace**, regardless of where it exists. It means that every other class can create objects from this type and has access to the methods and the fields of this public class.

Just to know, if we want to use a class with access level **public** from another namespace, we should use the reserved **using** word for including different namespaces, otherwise we always need to write the full name of the class.

### Access Level "internal"

If we declare a class with access modifier **internal**, it will be **accessible only from the same assembly**. It means that only the classes from the same assembly can create objects from an internal class and access the methods and fields (with related access level) of that class. This access level is the default when no access modifier is used in the declaration of the class.

If we have two projects in a solution in Visual Studio and a class from one project wants to use a class from the other project, then the latter referenced class should be declared as **public**.

## Access Level "private"

If we want to be exhaustive, we have to mention that the **private** access modifier for a class can be used, but this is related to the term "inner class" (nested class), which we will review in the "[Nested Classes](#)" section. Private classes, like other private members, are accessible only from inside the class which defines them.

## Body of the Class

By similarity to the methods, after the declaration of the class follows its body, i.e. the part of the class where resides the following programming code:

```
[<access_modifier>] class <class_name>
{
    // ... Class body - the code of the class goes here ...
}
```

The body of the class begins with an opening curly bracket "{" and ends with a closing one - "}". The class always should have a body.

## Class Naming Convention

Like for the methods, class naming follows common standards:

1. The names of the classes begin with a capital letter and the rest of the letters are lower case. If the name of the class consists of several words, each word begins with a capital letter, without using a separator. This is the well-known **PascalCase** convention.
2. **Nouns** are usually used for names of the classes.
3. It is recommended that the name of the class is in **English**.

Here are some example of class names, which are following the guidelines:

```
Dog
Account
Car
BufferedReader
```

We will learn more about naming of classes in the chapter "[High-Quality Programming Code](#)".

## The Reserved Word "this"

The reserved word **this** in C# is used to **reference the current object** from a method in the same class. The reserved word **this** is the object which method or constructor is called. The reserved word can be deemed as an address (reference), giving priority access to the elements (fields, methods, constructor) of the own class:

```
this.myField; // access a field in the class
this.DoMyMethod(); // access a method in the class
this(3, 4); // access a constructor with two int parameters
```

In the following sections of this chapter, dedicated to the elements of the class (fields, methods, constructors) we will further explain the use of the reserved word **this**.

## Fields

Objects describe things from the real world. In order to describe an object, we focus on its **characteristics**, which are related to the problems solved by our program. These characteristics of the real-world object we will hold in the declaration of the class in special types of variables. These variables, called **fields** (or member-variables), are holding the **state of the object**. When we create an object based on certain class definition, the values of the fields are containing the characteristics of the created object (its state). These characteristics have different values for the different objects.

## Declaring Fields in a Class

Until now we have discussed only two types of variables (see "[Methods](#)") depending on where they are declared:

1. **Local variables** – these are the variables declared in the body of some method (or code block).
2. **Parameters** – these are the variables in the list of parameters of a method.

In C#, a third type of variable exists, called a **field** or **instance variable**.

Fields are declared in the body of the class, but outside the body of methods and constructors.



**Fields are declared in the body of the class but not in the bodies of the methods or the constructors.**

This is a sample code declaring several fields:

```
class SampleClass
{
    int age;
    long distance;
    string[] names;
    Dog myDog;
}
```

More formal, a field is declared as follows:

```
[<modifiers>] <field_type> <field_name>;
```

The **<field\_type>** part determinates the type of a given field. This type can be primitive (**byte**, **short**, **char** and so on), an array, or also some class type (e.g. **Dog** or **string**).

The **<field\_name>** part is the name of the field. As for the names of normal variables, we name instance-variables according the naming rules for identifiers in C# (see chapter "[Primitive Types and Variables](#)").

The **<modifiers>** part is a definition, which describes the access modifiers, as well as other modifiers. Modifiers are not a mandatory part of the field declaration.

Modifiers in the declaration of a field are explained in chapter "[Primitive Types and Variables](#)".

In this chapter, besides access modifiers, we will discuss **static**, **const** and **readonly** modifiers, used in field declaration.

## Scope

The **scope of a class field** starts from the line where it is declared and ends at the closing bracket of the body of the class.

## Initialization during Declaration

When we declare a field, it is possible to assign an initial value, like assigning a value to a normal local variable:

```
[<modifiers>] <field_type> <field_name> = <initial_value>;
```

Of course, the **<initial\_value>** has to be of a type that is compatible with the field's type, e.g.:

```
class SampleClass
{
    int age = 5;
    long distance = 234; // The literal 234 is of integer type

    string[] names = new string[] { "Peter", "Martin" };
    Dog myDog = new Dog();

    // ... Other code ...
}
```

## Default Values of the Fields

When we create a new object of a class, memory is allocated in the heap for every field from the class. The memory is **initialized automatically with default values** for the various fields. The fields without explicitly defined default value in the code, use the default value specified by .NET for the type of the fields.

This is different for local variables defined in methods. If a local variable in a method does not have a value assigned, the code will not compile. If a member variable (field) in a class does not have a value assigned, it will be automatically zeroed by the compiler.



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

In some languages (C and C++) the newly created objects are not initialized with default values of their type and this creates conditions for hard-to-find errors, causing **uncontrolled behavior**. The program sometimes works correctly (when the allocated memory by chance has good values) and sometimes it does not work at all (when the allocated memory does not contain the proper values). In C# and generally in .NET Framework, this problem is solved by assigning default values for each type provided by the framework.

The value of all types is 0 or equivalent. For the most used types these values are as the follows:

Type of the Field	Default Value
bool	false

byte	0
char	'\0'
decimal	0.0M
double	0.0D
float	0.0F
int	0
object reference	null

For more detailed information, you can check chapter "[Primitive Types and Variables](#)" and its section about the [primitive types and their default values](#).

For example, if we create a class **Dog** and we define its fields **name**, **age**, **length** and **isMale**, without explicitly initializing them, then they will be automatically zeroed when we create an object of this class:

```
public class Dog
{
    string name;
    int age;
    int length;
    bool isMale;

    static void Main()
    {
        Dog dog = new Dog();
        Console.WriteLine("Dog's name is: " + dog.name);
        Console.WriteLine("Dog's age is: " + dog.age);
        Console.WriteLine("Dog's length is: " + dog.length);
        Console.WriteLine("Dog is male: " + dog.isMale);
    }
}
```

When we execute the program the output is as follows:

```
Dog's name is:
Dog's age is: 0
Dog's length is: 0
Dog is male: False
```

## Automated Initialization of Local Variables and Fields

If we define a local variable in a method without initializing, and afterward we try to use it (e.g. printing its value), then a **compilation error** is triggered because local variables are not initialized with default values when declared.



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

Let's have look at an example:

```
static void Main()
{
    int notInitializedLocalVariable;
    Console.WriteLine(notInitializedLocalVariable);
}
```

If we try to compile, we will receive the following error:

```
Use of unassigned local variable 'notInitializedLocalVariable'
```

## Custom Default Values

Good programming practice requires declaration of fields in the class followed by explicit initialization with a default value, even if the default value is zero. This will make our code clearer and easier to read.

An example of initialization is shown in the modified **SampleClass** of the [previous section](#):

```
class SampleClass
{
    int age = 0;
    long distance = 0;
    string[] names = null;
    Dog myDog = null;

    // ... Other code ...
}
```

## Modifiers "const" and "readonly"

As was explained in the beginning of this section, a field can be declared with the modifiers **const** and **readonly**. The fields declared with **const** or **readonly** are called **constants**. They are used when a certain **value is used several times**. These values are declared only once. Examples of constants in the .NET Framework are the mathematical constants **Math.PI** and **Math.E**, as well as the constants **String.Empty** and **Int32.MaxValue**.

### Constants Based on "const"

The fields declared with **const** have to be initialized during declaration and afterwards their value cannot be changed. They can be accessed without creating an instance (an object) of the class and they are shared by all objects created in our program. Additionally, when we compile the code, the **const** field reference is replaced by the const field value. Therefore, the **const** fields are called **compile-time constants**, because they are replaced with their value during compilation.

### Constants Based on "readonly"

The modifier **readonly** creates fields, which values cannot be changed once they are assigned. Fields declared as **readonly** allow one-time initialization, either the moment of declaration or after accessing class constructors. Thereafter, their values cannot be changed. Because of this, the **readonly** fields are called **run-time constants**. Constants, because their values

cannot be changed after assignment and run-time, because the one-time initialization occurs during execution of the program (during runtime).

Let's illustrate this with the following example:

```
public class ConstAndReadOnlyExample
{
    public const double PI = 3.1415926535897932385;
    public readonly double Size;

    public ConstAndReadOnlyExample(int size)
    {
        this.Size = size; // Cannot be further modified!
    }

    static void Main()
    {
        Console.WriteLine(PI);
        Console.WriteLine(ConstAndReadOnlyExample.PI);
        ConstAndReadOnlyExample instance =
            new ConstAndReadOnlyExample(5);
        Console.WriteLine(instance.Size);

        // Compile-time error: cannot access PI like a field
        Console.WriteLine(instance.PI);

        // Compile-time error: Size is instance field (non-
static)
        Console.WriteLine(ConstAndReadOnlyExample.Size);

        // Compile-time error: cannot modify a constant
        ConstAndReadOnlyExample.PI = 0;

        // Compile-time error: cannot modify a readonly
field
        instance.Size = 0;
    }
}
```

## Methods

In the chapter "[Methods](#)", we discussed how to **declare and use a method**. In this section we will revise how to do this and focus on some additional features of the process of creating methods. Now it is time to start using non-static (instance) methods.

## Declaring of Class Method

Methods are declared as follows:

```
// Method definition
[modifiers] [return_type] <method_name>([parameters_list])
```



```
{  
    // ... Method's body ...  
    [<return_statement>];  
}
```

The mandatory elements for declaration of a method are the type of the return value **<return\_type>**, the name of the method **<method\_name>** and the opening and the closing brackets – "(" and ")".

The parameter list **<params\_list>** is not mandatory. We use it to pass data to the method, when this is required.

If the return type **<return\_type>** is **void**, then **<return\_statement>** is omitted. If **<return\_type>** is different from **void**, then the method has to return a result with the help of the reserved word **return** and an expression, which is from the same type as **<return\_type>** or of a compatible type.

The work, which the method has to do, is defined in the method body, enclosed in curly brackets – "{" and "}".

We will revise some of the access modifiers that can be used in the declaration of a method, already discussed in the section "[Visibility of Methods and Fields](#)".

The **static** modifier will be explained in depth in the section "[Static Classes and Static Members](#)".

### Example – Method Declaration

Let's consider the declaration of a method, which sums two values:

```
int Add(int number1, int number2)  
{  
    int result = number1 + number2;  
    return result;  
}
```

The name of the method is **Add** and the return value type is **int**. The parameter list consists of two elements – the variables **number1** and **number2**. Accordingly, the return value is the sum of the two parameters as a result.

## Accessing Non-Static Data of the Class

In "[Creating and Using Objects](#)", we have discussed how the "dot" operator allows accessing fields and calling methods of a class. Now, let's recall how we use conventional non-static methods of a class, i.e. the methods without the modifier **static** in their declaration.

Consider the class **Dog** with the field **age**. To print the value of this field we need to create a **Dog** instance and then access the field of this instance via a "dot" notation:

```
public class Dog  
{  
    int age = 2;  
  
    static void Main()  
}
```

```
{
    Dog dog = new Dog();
    Console.WriteLine("Dog's age is: " + dog.age);
}
```

The result will be:

```
Dog's age is: 2
```

## Accessing Non-Static Fields from Non-Static Method

The value of a field can be accessed via the "dot" notation (as in the last example `dog.age`), or via a method or property. Now, let's create in the class `Dog` a method, which will return the value of `age`:

```
public int GetAge()
{
    return this.age;
}
```

As we see, to access the value of the `age` field inside the owner class, we use **the reserved word `this`**. We know that the word **`this`** is a reference to the current object, in which the method resides. Therefore, in our example with "`return this.age`" we say "from the current object (**`this`**), take (the use of the operator "dot"), the value of the field `age`, and return it as result from the method (with the help of the reserved word `return`). Then, instead of accessing the value of the field `age` of the object `dog` from the `Main()` method, we simply call the method `GetAge()`:

```
static void Main()
{
    Dog dog = new Dog();
    Console.WriteLine("Dog's age is: " + dog.GetAge());
}
```

The result of the execution after the change will be the same.

Formally, the declaration of access to a field within the boundaries of a class is the following:

```
this.<field_name>
```

Let's emphasize that this access option is possible only from non-static code, i.e. method or code block, which is thus without the **`static`** modifier.

Besides for retrieval of the value of a field, we can use the reserved word **`this`** for modification of the field.

E.g., let's declare a method `MakeOlder()`, which will be called every year on the date of the birthday of our pet, and then increments the age with one year:

```
public void MakeOlder()
{
    this.age++;
}
```

To check if this is correct, we add the following lines to the **Main()** method:

```
// One year later, at the birthday date...
dog.MakeOlder();
Console.WriteLine("After one year dog's age is: " + dog.age);
```

After the execution of the program, the result is the following:

```
Dog's age is: 2
After one year dog's age is: 3
```

## Calling Non-Static Methods

Like the fields, which do not have a **static** modifier in their declaration, the methods, which are also non-static, can be called in the body of a class via the reserved word **this**. This is done again with the "dot" notation and with any required arguments:

```
this.<method_name>(...)
```

For example, let's create a method **PrintAge()**, which has to print the age of the object from type **Dog** and therefore calls the method **GetAge()**:

```
public void PrintAge()
{
    int myAge = this.GetAge();
    Console.WriteLine("My age is: " + myAge);
}
```

The first line of the code between curly brackets indicates that we want to retrieve the age (the value of the field **age**) from the current object, using the method **GetAge()**. This is done via the reserved word **this**.



**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".**

## Omit "this" Keyword When Accessing Non-Static Data

When we access the fields of a class or we call its non-static methods, it is possible to **omit the reserved word `this`**. Then, both methods, which we already declared, will be written in this way:

```
Public int GetAge()
{
    return age; // The same as this.age
}
```

```

}

public void MakeOlder()
{
    age++; // The same as this.age++
}

```

The reserved word **this** is used to indicate **explicitly** that we want to have access to a non-static field of a class or to call some of its non-static methods. When this explicit clarification is not needed, it can be omitted to access the elements of the class directly.

Nevertheless, the reserved word **this** is often used for the access of fields in the class, because it makes the code easier to read, understand and maintain, by explicitly stating that we access a field and not a local variable.



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

## Hiding Fields with Local Variables

From the section "[Declaring Fields](#)" above, we know that the **scope of a field** is from the line where the declaration is made to the closing curly bracket of the class. For example, let's consider the **OverlappingScopeTest** class:

```

public class OverlappingScopeTest
{
    int myValue = 3;

    void PrintMyValue()
    {
        Console.WriteLine("My value is: " + myValue);
    }

    static void Main()
    {
        OverlappingScopeTest instance = new
OverlappingScopeTest();
        instance.PrintMyValue();
    }
}

```

This code will have the following result on the console:

```
My value is: 3
```

When we implement the body of a method we may have to declare local variables which we will use in the method. As we know, the **scope of a local variable** in a method is from the line where it is declared to the closing bracket of the body of the method. For example, let's add the following method to the class **OverlappingScopeTest**:

```
Int CalculateNewValue(int newValue)
{
    int result = myValue + newValue;
    return result;
}
```

In this case, the local variable, which we will use to calculate the new value, is **result**.

Sometimes the name of the local variable can have the same name as a field. In that case, there is a conflict.

Let's first look at an example, before we explain what it is all about. Let's modify the method **PrintMyValue()** in the following way:

```
void PrintMyValue()
{
    int myValue = 5;
    Console.WriteLine("My value is: " + myValue);
}
```

If we declare the method in this way, then is it possible to compile this code? Is it possible to execute it and if so, which value will be printed – the one of the field or the one of the local variable?

After the execution of the **Main()** method, the result will be:

```
My value is: 5
```

This is so, because **C# allows defining local variables, which have the same name as the fields of the class**. In such case, we say that the scope of the local variable overlaps the field variable (**scope overlapping**).

Therefore, the scope of the local variable **myValue** with value **5** overlaps the scope of the field variable in the class. When we print it, we get the value of the local variable.

Sometimes, it is required to use the field instead of the local variable with the same name. In this case, we retrieve the value of the field by using the reserved word **this** and access the field by using the "dot" operator. In this way, we say explicitly that we want to use the field of the class and not the local variable with the same name.

Let's take a look again at our example and the printing of **myValue**:

```
void PrintMyValue()
{
    int myValue = 5;
    Console.WriteLine("My value is: " + this.myValue);
}
```

This time, after applying the changes, the result from the call of the method is different:

```
My value is: 3
```

## Visibility of Fields and Methods

In the beginning of this chapter, we have discussed the **modifiers and the access levels** for the elements in a class in C# and the access level in the declaration of a class.

Now we will discuss the **visibility levels of fields and methods** in a class. Because the fields and the methods are elements of the class (members) and have similar rules for access levels, we will explain these rules simultaneously.

Different from the declaration of a class, when we declare fields and methods in the class, we can use the four access levels – **public**, **protected**, **internal** and **private**. The access level **protected** will not be discussed in this chapter, because it is related to class inheritance and is explained in details in the chapter "[Object-Oriented Programming Principles](#)".

Before we continue, let's revise. If one class **A** is not visible (does not give access) from another class **B**, then none of its elements (fields and methods) can be accessed from class **B**.



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

In the next subsections, we will review examples, of two classes (**Dog** and **Kid**) which are visible to each other. Each class can create objects of the other class type and access its elements depending on their declared access levels. Here is how the first class **Dog** looks like:

```
public class Dog
{
    private string name = "Doggy";

    public string Name
    {
        get { return this.name; }
    }

    public void Bark()
    {
        Console.WriteLine("wow-wow");
    }

    public void DoSomething()
    {
        this.Bark();
    }
}
```

In addition to the fields and the methods, the property **Name** is used, which just returns the field's value. We will discuss in details the property concept later, so currently, we will just focus on everything else except properties.

The code of the class **Kid** looks like this:

```

public class Kid
{
    public void CallTheDog(Dog dog)
    {
        Console.WriteLine("Come, " + dog.Name);
    }

    public void WagTheDog(Dog dog)
    {
        dog.Bark();
    }
}

```

For the moment, all elements (fields and methods) of both classes are declared with the access modifier **public**. We like to find out how changing the access levels of the elements (fields and methods) of the class **Dog** will affect access from:

- The own body of the class **Dog**.
- The body of the class **Kid**, taking into account that **Kid** is either in the same or in a different namespace (assembly) as is the **Dog** class.

## Access Level "public"

When methods or values of a class are declared with access level **public**, then they **can be accessed from other classes**, independent from declaration in the same or a different namespace.

Let's review the types of access to members of a class, as with our classes **Dog** and **Kid**:

<b>D</b>	The member of the class is accessed inside the same class <b>directly</b> (the class refers itself).
<b>R</b>	The member of the class is accessed via a <b>reference</b> to an object created from another class (the class refers another class).

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

Dog.cs	
<b>D</b>	<pre> class Dog {     public string name = "Doggy";     public string Name     {         get { return this.name; }     }      public void Bark()     { </pre>

D	<pre>         Console.WriteLine("wow-wow");     }     public void DoSomething()     {         this.Bark();     } } </pre>
---	---------------------------------------------------------------------------------------------------------------------------

Kid.cs	
R	<pre> class Kid {     public void CallTheDog(Dog dog)     {         Console.WriteLine("Come, " + dog.name);     }     public void WagTheDog(Dog dog)     {         dog.Bark();     } } </pre>
R	

As we can see, we implement without problem access to the field **name** and the method **Bark()** of the class **Dog** from the body of the same class. From another class **Kid**, in the same namespace as **Dog**, we access the field **name** and call the method **Bark()** via the "dot" operator and the reference to the object of type **Dog**.

## Access Level "internal"

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

Note that if we have a Visual Studio project, all classes in it are from the same assembly and classes defined in different Visual Studio projects (in the same or in a different solution) are from different assemblies.

Below is the explanation of the access level **internal**:

Dog.cs	
	<pre> class Dog {     internal string name = "Doggy";      public string Name     {         get { return this.name; }     } } </pre>



D	<pre>     }      internal void Bark()     {         Console.WriteLine("wow-wow");     }      public void DoSomething()     {         this.Bark();     } </pre>
D	}

For the classes Dog and Kid, we discuss two cases:

- When the classes are in **the same assembly**, then the access to the elements of **Dog** from Kid will be allowed, regardless whether the classes are in the same namespace or not:

Kid.cs	
R	<pre> class Kid {     public void CallTheDog(Dog dog)     {         Console.WriteLine("Come, " + dog.name);     } </pre>
R	<pre>     public void WagTheDog(Dog dog)     {         dog.Bark();     } } </pre>

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

Kid.cs	
<del>R</del>	<pre> class Kid { <del>    public void CallTheDog(Dog dog)</del> <del>    {</del> <del>        Console.WriteLine("Come, " + dog.name);</del> <del>    }</del>  <del>    public void WagTheDog(Dog dog)</del> <del>    {</del> <del>        dog.Bark();</del> <del>    }</del> } </pre>

	} }
-----------------------------------------------------------------------------------	--------



Actually, access to **internal** members of the class **Dog** is impossible for two reasons: insufficient visibility of the class and insufficient visibility of its members. To allow access from another assembly to the class **Dog**, both, the class and its members, have to be declared **public**. If the class or its members have lower visibility, access from other assemblies is denied (i.e. from other Visual Studio projects which compile to different **.dll** / **.exe** files).

If we try to compile the class **Kid**, when it is external to the assembly in which the class **Dog** resides, we will get a compilation error.

## Access Level "private"



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

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 an instance of the same class **Dog**, while access from any other outside classes is not permitted. If you try to access a private method from an external class, a compilation error occurs. Below is the figure for the access level **private**:

Dog.cs	
	<pre> class Dog {     private string name = "Doggy";      public string Name     {         get { return this.name; }     }      private void Bark()     {         Console.WriteLine("wow-wow");     }      public void DoSomething()     {         this.Bark();     } } </pre>
	

Accessing the **name** fields from the same class is permitted, while accessing them from a different class (**Kid**) is not:

Kid.cs
--------

	<pre> class Kid {     <del>public void</del> CallTheDog(Dog dog)     {         Console.WriteLine("Come, " + dog.name);     }      <del>public void</del> WagTheDog(Dog dog)     {         dog.Bark();     } } </pre>
	

When we assign an access modifier to a field, then in most of the cases it better is **private**, because this ensures the highest level of security. Access to, and modification of, the value by other classes will occur via properties or methods. More about this technique in the section ["Properties and Encapsulation of Fields"](#), as well as in the ["Encapsulation"](#) section of the chapter ["Object-Oriented Programming Principles"](#).

## How to Decide Which Access Level to Use?

Before we end the section regarding visibility of the elements of a class, let's define in the class **Dog** the field **name** and the method **Bark()** with access modifier **private** and declare a method **Main()**:

```

public class Dog
{
    private string name = "Doggy";

    // ...

    private void Bark()
    {
        Console.WriteLine("wow-wow");
    }

    // ...

    static void Main()
    {
        Dog myDog = new Dog();
        Console.WriteLine("My dog's name is " + myDog.name);
        myDog.Bark();
    }
}

```

The question is, whether the class **Dog** compiles when we declare elements with access modifier **private** and try to access them by means of a "dot" notation to **myDog** in **Main()**?

The **compilation finishes successfully** and the result from the execution of the method **Main()** will be the following:

```
My dog's name is Rolf  
Wow-wow
```

Everything works fine. As the variable **myDog** is defined in the body of the class **Dog**, like the **Main()** method – the start of the program, we can access its elements (fields and methods) via “dot” notation, even after we declared the access level as **private**. However, it is not possible to access the same from the body of the class **Kid**, because the access to **private** fields from outside of the class **Dog** is forbidden.

## Constructors

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?

The **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 non-**static**).

### Calling a Constructor

The only way to **call a constructor** in C# is through the **keyword new**. It allocates memory for a 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.

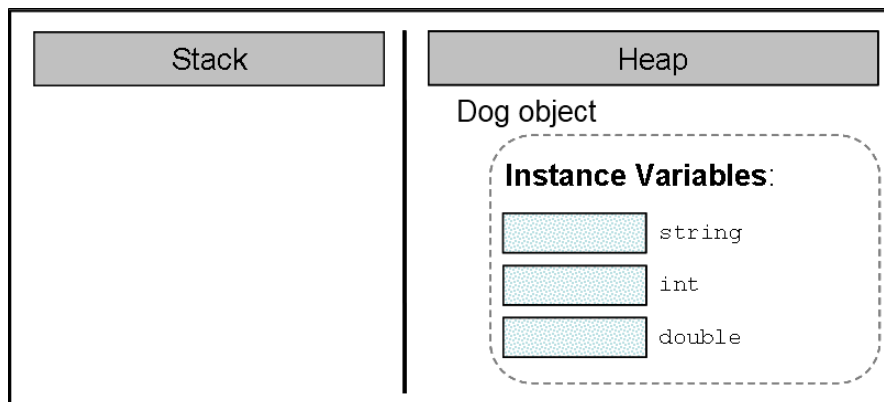
Consider an example, which will clarify how the constructor works. We know from chapter ["Creating and Using Objects"](#) how to create an object:

```
Dog myDog = new Dog();
```

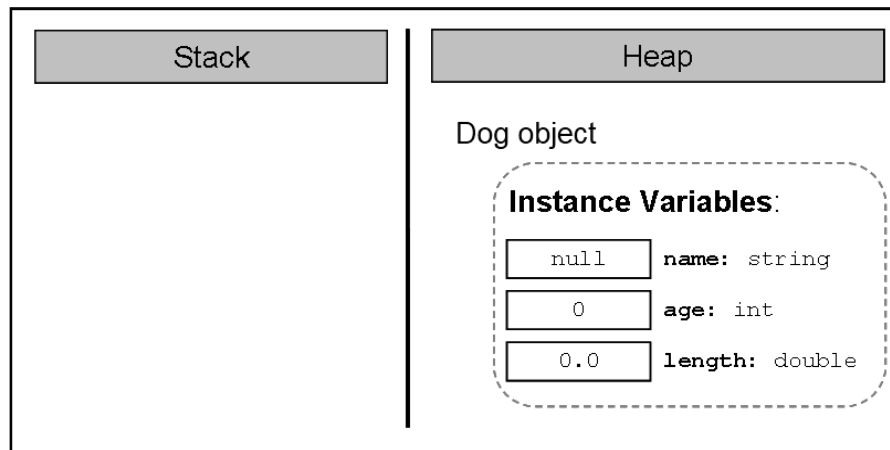
In this case, by using the keyword **new** we call the constructor of the class **Dog** and by doing so, memory is allocated, needed for the newly created object of the **Dog** type. When it comes to classes, they are allocated in the dynamic memory (in the so called "managed heap").

Let's follow the process of calling a constructor during the creation of a new object step by step.

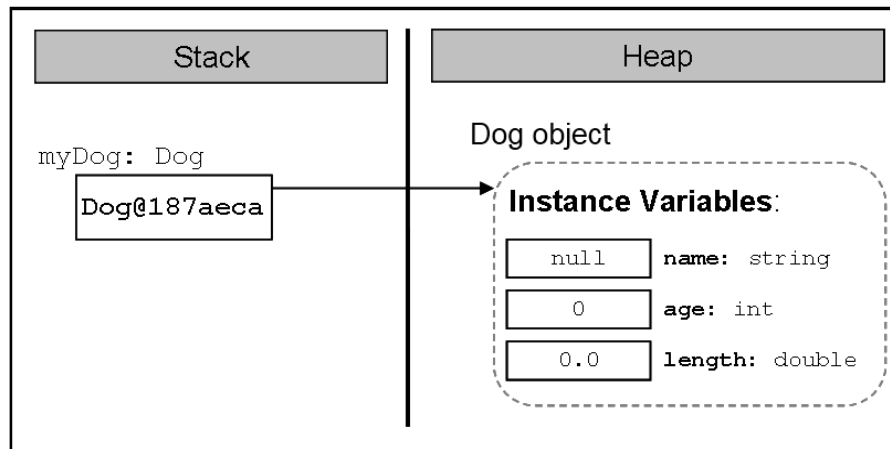
First, **memory is allocated** for the object:



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



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**:



## Declaring a Constructor

If we have the class **Dog**, here is how its most simplified constructor (without parameters) will look like:

```
public Dog()  
{  
}
```

Formally, the declaration of the constructor appears in the following way:

```
[<modifiers>] <class_name>([<parameters_list>])
```

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

## Constructor's Name

In C# it is mandatory that **the name of every constructor is the same as the name of the class in which it resides** - `<class_name>`. In the example above, the name of the

constructor is the same as the name of the class – **Dog**. We should know that, as with methods, the name of the constructor is always followed by round brackets – "(" and ")".

In C# it **is not allowed to declare a method whose name matches the name of the class** (name reserved for the constructors). If, nevertheless, a method is declared with the class name, this will cause a compilation error.

```
public class IllegalMethodExample
{
    // Legal constructor
    public IllegalMethodExample ()
    {
    }

    // Illegal method
    private string IllegalMethodExample()
    {
        return "I am illegal method!";
    }
}
```

When we try to compile this class, the compiler will display the following **compilation error message**:

```
SampleClass: member names cannot be the same as their enclosing
type
```

## Parameter List

Similar to the methods, if we need extra data to create an object, the constructor gets these through a **parameter list** – **<parameters\_list>**. In the example constructor of the class **Dog** there is no need for additional data to create an object of this type and therefore there is no parameter list. More about the parameter list will be explained in one of the later sections – "[Declaring a Constructor with Parameters](#)".

Of course, after the declaration of the constructor, its body is following, which is like every method body in C#, but generally contains mostly initialization logic, i.e. setting the initial values of the fields of the class.

## Modifiers

**Modifiers** can be added to the declaration of the constructors – **<modifiers>**. For the discussed modifiers, which are not access modifiers, i.e. **const**, **readonly** and **static**, only **static** is allowed in the declaration of constructors. Later in this chapter, in the "[Static Constructors](#)" section, we will learn more about the constructors declared with modifier **static**.

## Visibility of the Constructors

Similar to the methods and the fields, the constructors can be declared with **levels of visibility**: **public**, **protected**, **internal**, **protected internal** and **private**. The access levels **protected** and **protected internal** will be explained in chapter "[Object-Oriented](#)

[Programming Principles](#)". The rest of the access levels have the same meaning and behavior as for fields and methods.

## Initialization of the Fields in the Constructor

As explained earlier, when creating a new object and calling its constructor, 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 (see the section "[Calling a Constructor](#)").

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

E.g., in the examples we discussed so far, the field **name** of the object of type **Dog** is always initialized during its declaration:

```
string name = "Sharo";
```

Instead of initializing during the declaration of the field, a better programming style is to assign a 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 declaration with the purpose of improving the readability of the code, but that is considered a matter of personal choice:

```
public class Dog
{
    private string name = null;

    public Dog()
    {
        this.name = "Sharo";
    }

    // ... The rest of the class body ...
}
```

## Fields Initialization in the Constructor

Let's see in detail what the constructor does after being called and the class fields have been initialized in its body. We know that, when called, it will **allocate memory** for each field and this **memory will be initialized** with the default values.

If the fields are of primitive type, then after the default values, we shall assign new values.

In case the fields are from reference type, such as our field **name**, the constructor will initialize them with **null**. It will then create the object of the corresponding type, in this case the string **"Sharo"** and at the end a reference will be assigned to the new object in the respective field, in our case the field **name**.

The same will happen if we have other fields, which are not primitive types, and then initialize them in the constructor. E.g. let's have a class called Collar, which describes a dog's accessory – **Collar**:

```
public class Collar
{
    private int size;

    public Collar()
    {
    }
}
```

Let our class **Dog** have a field called **collar**, which is from type **Collar** and which is initialized in the constructor of the class:

```
public class Dog
{
    private string name;
    private int age;
    private double length;
    private Collar collar;

    public Dog()
    {
        this.name = "Sharo";
        this.age = 3;
        this.length = 0.5;
        this.collar = new Collar();
    }

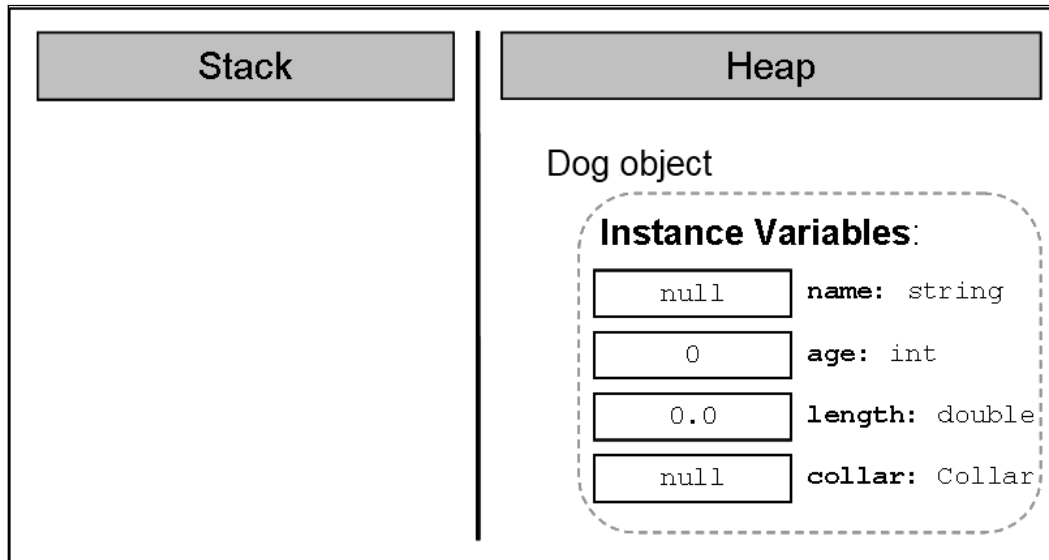
    static void Main()
    {
        Dog myDog = new Dog();
    }
}
```

## Representation in the Memory

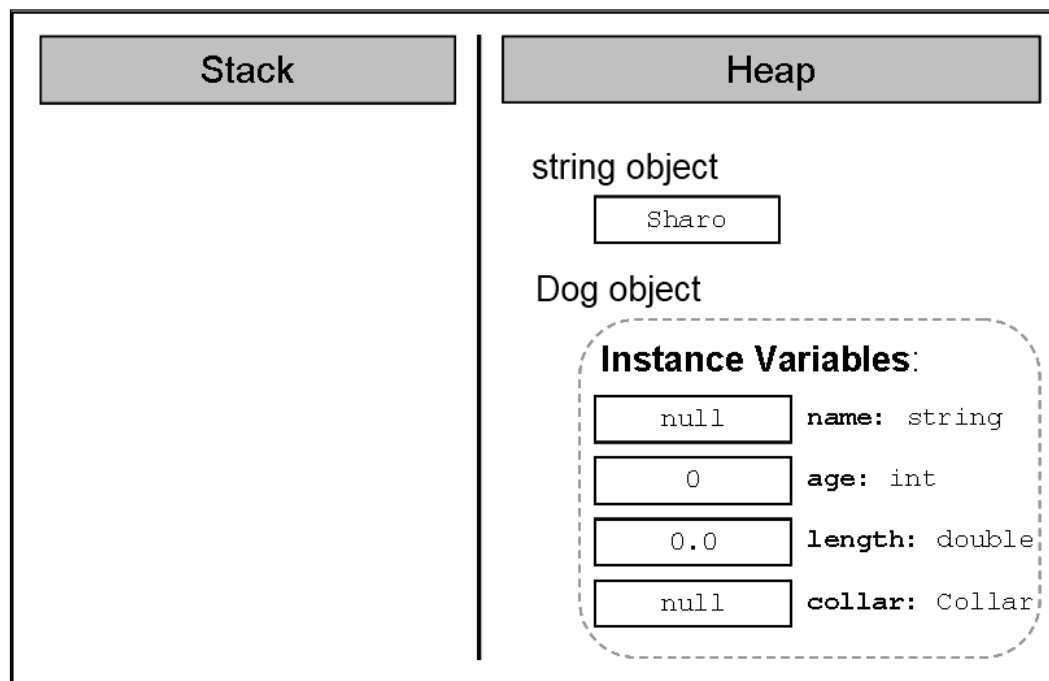
Let's follow the steps through which the constructor goes, after being called in the **Main()** method.

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:

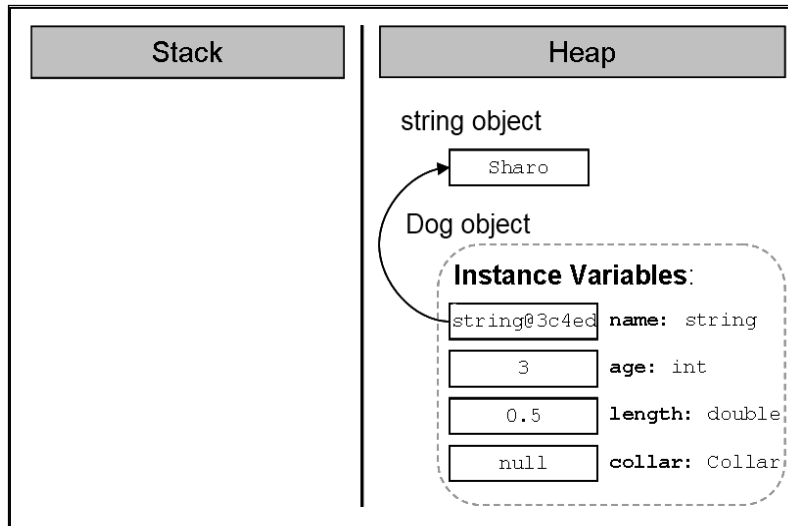




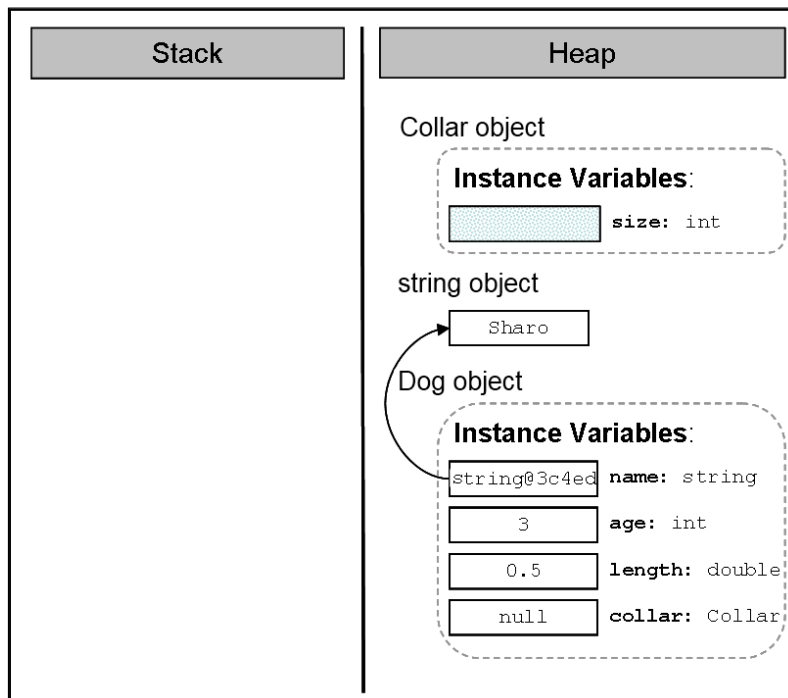
Then, the constructor will have to ensure the creation of the object for the field **name**. It will **call the constructor of the class string**, which will do the work on the string creation:



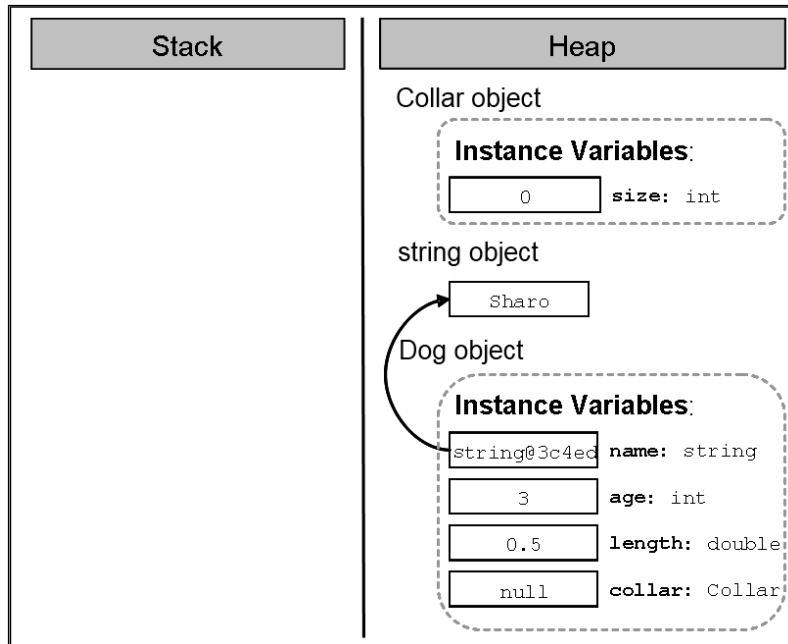
Now the constructor will keep the reference to the new string in the field **name** of the **Dog** object:



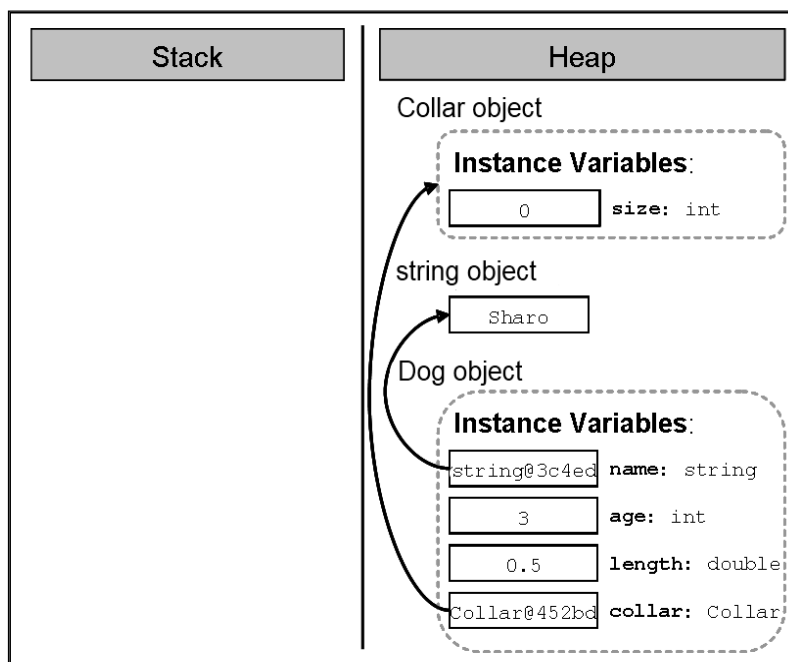
Then follows the creation of the object from type **Collar**. Our constructor (of the class **Dog**) calls the constructor of the class **Collar**, which allocates memory for the object:



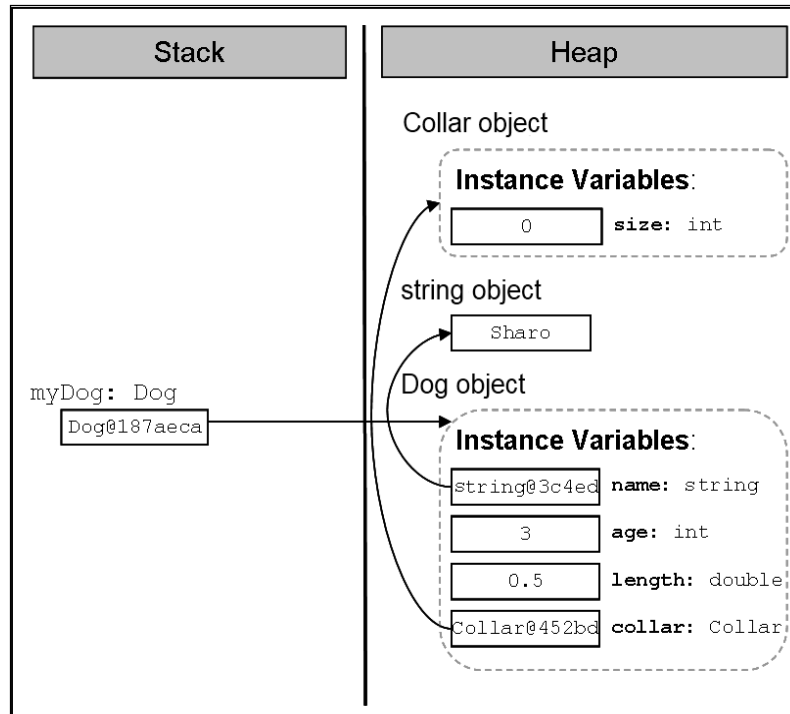
Next, the constructor will **initialize its field, size, with the default value** for the respective type. The **size** of the **Collar** is not explicitly assigned, so it will take the default value for its type (**0** for **int**):



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**:



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



## Order of Initialization of the Fields

To avoid confusion, let's explain the **order in which the fields of a class are initialized** regardless of whether we have assigned to them values and / or initialized them in the constructor.

First, **memory is allocated** for the respective field in the heap and this memory is **initialized** with the default value of the field type. E.g. let's again consider the example with the class **Dog**:

```
public class Dog
{
    private string name;

    public Dog()
    {
        Console.WriteLine(
            "this.name has value of: \"" + this.name +
            "\"");
        // ... No other code here ...
    }
    // ... Rest of the class body ...
}
```

When we try to create a new object of our class type, the console will show:

```
this.name has value of: ""
```

After the initialization of the fields with the default value of the respective type, the second step of the CLR (Common Language Runtime) is to **assign a value to the field**, if such has been set when declaring the field.

So, if we change the line in the class **Dog**, where we declare the field **name**, it will first be initialized with the value **null** and then it will be assigned the value **"Rex"**.

```
private string name = "Rex";
```

Respectively, for every creation of a new object from the class **Dog**:

```
static void Main()
{
    Dog dog = new Dog();
}
```

The following will be printed:

```
this.name has value of: "Rex"
```

Only after these two steps of initializing the fields of the class (default value initialization and eventual initialization of the value set by the programmer during the declaration of the field) **the constructor of the class is called**. At this time, the fields get the values, which are set in the body of the constructor.

## Declaring a Constructor with Parameters

In the previous section, we saw how we can set non-default values of fields. However, during the declaration of the constructor, we often don't know what values the various fields will have. Therefore, **similar to methods with parameters**, fields are assigned values, provided in the body of the constructor. For example:

```
public Dog(string dogName, int dogAge, double dogLength)
{
    name = dogName;
    age = dogAge;
    length = dogLength;
    collar = new Collar();
}
```

Similarly, we **call a constructor with parameters** in the same way as calling a method with parameters – the required values are supplied as a list, the elements of which are separated with commas:

```
static void Main()
{
    Dog myDog = new Dog("Moby", 2, 0.4); // Passing parameters

    Console.WriteLine("My dog " + myDog.name +
        " is " + myDog.age + " year(s) old. " +
        " and it has length: " + myDog.length + " m.");
}
```

The result of the execution of this **Main()** method is:

```
My dog Moby is 2 year(s) old. It has length: 0.4 m.
```

There is no limitation to the number of constructors of a class in C#. The only requirement is that they **differ in their signature** (what signature is, we already explained in chapter ["Methods"](#)).

### Scope of Parameters of the Constructor

By analogy with the scope of the variables in the parameter list of a method, the **variables in the parameter list of a constructor have a scope** from the opening bracket of the constructor to the closing bracket, i.e. throughout the body of the constructor.

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**:

```
public Dog(string name, int age, double length)
{
    name = name;
    age = age;
    length = length;
    collar = new Collar();
}
```

Let's compile and execute the [Main\(\) method declared above](#):

```
My dog is 0 year(s) old. It has length: 0 m
```

Unexpected result, isn't it? In fact, this result is not so strange. The explanation is the following: the scope of the variables from the parameter list of the constructor overlaps the scope of the fields in the constructor, with the same names. Thus, **we did not assign any value to the fields** because we had no access to them. For example, instead of assigning the variable value to the field **age**, we assigned it to the **age** variable itself:

```
age = age;
```

As we saw in the section ["Hiding Fields with Local Variables"](#), to avoid this problem we should access the field, to which we want to assign a value, by **using the keyword this**:

```
public Dog(string name, int age, double length)
{
    this.name = name;
    this.age = age;
    this.length = length;
    this.collar = new Collar();
}
```

Now, assuming we execute again the **Main()** method:

```
static void Main()
```

```

{
    Dog myDog = new Dog("Moby", 2, 0.4);

    Console.WriteLine("My dog " + myDog.name +
        " is " + myDog.age + " year(s) old. " +
        " and it has length: " + myDog.length + " m");
}

```

The result will be exactly what we expect it to be:

```
My dog Moby is 2 year(s) old. It has length: 0.4 m
```

## Constructor with Variable Number of Arguments

Similar to methods with **variable number of arguments**, discussed in chapter ["Methods"](#), constructors can also be declared with a parameter for a variable number of arguments. The rules for declaring and calling constructors with a variable number of arguments are the same as the ones described for declaring and calling methods:

1. When we declare a constructor with a variable number of arguments, we must use **the reserved word params**, and then insert the type of the parameters, followed by square brackets. Finally, the name of the array follows, in which 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 in the parameter list.
3. The parameter for variable number of arguments must be the last in the parameter list of the constructor.

Consider an **example declaration** of a constructor of a class, which describes a lecture:

```

public Lecture(string subject, params string[] studentsNames)
{
    // ... Initialization of the instance variables ...
}

```

The first parameter in the declaration is the name of the subject of the lecture and the next parameter represents a **variable number of arguments** – the names of the students. Here is how an object of this class would be constructed:

```

Lecture lecture =
    new Lecture("Biology", "Peter", "Mike", "Steven");

```

The first parameter is the name of the subject – **"Biology"**, and all the other arguments are the names of the attending students.

## Constructor Overloading

Declaring constructors with parameters gives us a possibility to declare constructors with different signatures (number and order of the parameters), for more convenient creation of

objects from our class. Defining **constructors with different signatures** is called **constructor overloading**.

Consider for example the class **Dog**. We can declare different constructors:

```
// No parameters
public Dog()
{
    this.name = "Ax1";
    this.age = 1;
    this.length = 0.3;
    this.collar = new Collar();
}

// One parameter
public Dog(string name)
{
    this.name = name;
    this.age = 1;
    this.length = 0.3;
    this.collar = new Collar();
}

// Two parameters
public Dog(string name, int age)
{
    this.name = name;
    this.age = age;
    this.length = 0.3;
    this.collar = new Collar();
}

// Three parameters
public Dog(string name, int age, double length)
{
    this.name = name;
    this.age = age;
    this.length = length;
    this.collar = new Collar();
}

// Four parameters
public Dog(string name, int age, double length, Collar collar)
{
    this.name = name;
    this.age = age;
    this.length = length;
    this.collar = collar;
}
```



## Reusing Constructors

In above example we see that, depending on the need for creating objects of our class, we can declare different variants of the constructors. One notices that a large part of the **constructor code is repeated**. This raises the question whether there is an alternative way for an initializing constructor, to be reused to perform the same initialization again.

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

```
[<modifiers>] <class_name>(<parameters_list_1>))  
    : this(<parameters_list_2>))
```

To the familiar declaration of a constructor (the first line of the declaration above), we add a colon followed by the keyword **this** with parentheses. For the constructor with parameters, a list of parameters (**parameters\_list\_2**) is to be supplied between brackets.

Here is how the code from the [section about constructor overloading](#) would look like, when we call the constructors declared in the same class without re-initializing each field:

```
// No parameters  
public Dog()  
    : this("Ax1") // Constructor call  
{  
    // More code could be added here  
}  
  
// One parameter  
public Dog(string name)  
    : this(name, 1) // Constructor call  
{  
}  
  
// Two parameters  
public Dog(string name, int age)  
    : this(name, age, 0.3) // Constructor call  
{  
}  
  
// Three parameters  
public Dog(string name, int age, double length)  
    : this(name, age, length, new Collar()) // Constructor  
call  
{  
}  
  
// Four parameters  
public Dog(string name, int age, double length, Collar collar)  
{  
    this.name = name;  
    this.age = age;
```

```
this.length = length;  
this.collar = collar;  
}
```

As commented in the first constructor, in addition to calling any of the other constructors with certain parameters, every constructor can add into its body code, which performs additional initializations or other actions.

## Default Constructor

What happens if we don't declare a constructor in our class? How can we create objects of that type?

If a class is without a single constructor, C# resolves the issue automatically. When we do not declare any constructors, the compiler will create one for us and this one will be used to create objects from our class. This constructor is called the **default implicit constructor**. It is without parameters and will be empty (i.e. it will not do anything but default zeroing of object fields).



**When we do not declare any constructor in a given class, the compiler will create one, known as a default implicit constructor.**

For example, let's declare the class **Collar**, without declaring any constructor:

```
public class Collar  
{  
    private int size;  
  
    public int Size  
    {  
        get { return size; }  
    }  
}
```

Although we do not have an explicitly declared constructor, we can create objects of this class in the following way:

```
Collar collar = new Collar();
```

The **default parameterless constructor** looks like:

```
<access_level> <class_name>() { }
```

We should know that the default constructor is always named like the class **<class\_name>**, and its parameter list and body are always empty. The default constructor is usually **public** (except for some very specific situations, where it is **protected**).



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

To demonstrate that the default constructor is always without parameters, let's try to call the default constructor with a parameter:

```
Collar collar = new Collar(5);
```

The compiler will display the following error message:

```
'Collar' does not contain a constructor that takes 1 arguments
```

## How the Default Constructor Works?

The only thing the default constructor will do when creating objects from our class, is to zero the fields of the class. For example, if in the class **Collar** we have not declared any constructor and we create an object from it, trying to print the value of the field **size**:

```
static void Main()
{
    Collar collar = new Collar();
    Console.WriteLine("Collar's size is: " + collar.Size);
}
```

Then the result will be:

```
Collar's size is: 0
```

We see that the value saved in the field **size** of the object **collar** is just the default value of the whole number type – **int**.

## When Will a Default Constructor Not Be Created?

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

To demonstrate this, consider the following example:

```
public Collar(int size)
    : this()
{
    this.size = size;
}
```

Let this be **the only constructor in the class Collar**. We try to call another constructor without parameters, hoping that the compiler will have created a default parameterless constructor for us. However, after we try to compile, we will find out that what we are trying to do is not possible. The compiler will show the following error:

```
'Collar' does not contain a constructor that takes 0 arguments
```

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

Before we finish this section on the constructors, we have to clarify something very important:



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

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, the default constructor always has either access level **protected** or **public**, depending on the access modifier of the class; while the level of access of the non-default constructor without parameters is defined by us.

## Properties

In the world of object-oriented programming, there is an element of the classes called **property**, which is **somewhere between a field and a method** and serves to better protect the state of the class. In many languages for object-oriented programming, like C#, Delphi / Pascal, Visual Basic, Python, JavaScript, and others, the properties are a part of the language, i.e. there is a special mechanism to declare and use them. Other languages like Java do not support the property concept and therefore, programmers have to provide this functionality by declaring a pair of methods for reading and modifying state.

## Properties in C# – Introduction by Example

Using properties is a proven best practice and important part of the concept of object-oriented programming. A property is created by **declaring two methods** – one for reading (**getting**) and one for modifying (**setting**) the value of the respective property.

Consider again class **Dog**, which describes a dog. A characteristic of a dog is, for example, its color. Reading of the property "color" of a dog and its modification can be accomplished in the following way:

```
// Getting (reading) a property
string colorName = dogInstance.Color;

// Setting (modifying) a property
dogInstance.Color = "black";
```

## Properties – Encapsulation of Fields

The main objective of the properties is to ensure the **encapsulation of the state of the class**, i.e. to protect the class from falling into **invalid state**.

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

Through the C# syntax, this is done by declaring the fields (physical presentation of data) with the most limited level of visibility possible (generally with the modifier **private**) and making sure that access to these fields (reading and modifying) is possible only through special **accessor methods**.

## Example of Encapsulation

To illustrate what encapsulation is and what the provided properties themselves represent, we consider an example.

Let's have a class which defines a **point in 2D space** with properties representing the coordinates {x, y}. Here is how it would look if we declare each of the coordinates as a field:

Point.cs
<pre>class Point {     private double x;     private double y;      public Point(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; }     } }</pre>

The fields of our class (i.e. the point's coordinates) are declared as **private** and cannot be accessed by a "dot" notation. If we create an object from class **Point**, we can modify and access the coordinates of the point only through the properties:

PointTest.cs
<pre>using System;  class PointTest {     static void Main()     {         Point myPoint = new Point(2, 3);          double myPointXCoord = myPoint.X; // Access a property</pre>

```

        double myPointYCoord = myPoint.Y; // Access a
property
        Console.WriteLine("The X coordinate is: " +
myPointXCoord);
        Console.WriteLine("The Y coordinate is: " +
myPointYCoord);
    }
}

```

The result of the execution of the **Main()** method will be:

```

The X coordinate is: 2
The Y coordinate is: 3

```

If we decide to change the internal representation of the point's properties, e.g. instead of two fields, we declare them as a one-dimensional array with two elements; we can do this without affecting the functioning of other classes in our project:

#### Point.cs

```

using System;

class Point
{
    private double[] coordinates;

    public Point(int xCoord, int yCoord)
    {
        this.coordinates = new double[2];

        // Initializing the x coordinate
        coordinates[0] = xCoord;

        // Initializing the y coordinate
        coordinates[1] = yCoord;
    }

    public double X
    {
        get { return coordinates[0]; }
        set { coordinates[0] = value; }
    }

    public double Y
    {
        get { return coordinates[1]; }
        set { coordinates[1] = value; }
    }
}

```

```
}
```

The result of the implementation of the **Main()** method will be the same without changing even a single character in the code of the class **PointTest**.

This demonstration is a **good example of data encapsulation** of an object, provided by the mechanism of properties. Properties **hide the internal representation** of the information. By declaring properties and methods for access, a change of representation will not affect other classes that use our class, because they only use properties and do not know how the information is represented "behind the scene".

Of course, the example shows only one of the benefits of wrapping (packing) the class fields into properties. **Properties allow further control over the data** in the class, as they can check whether the assigned values are correct according to some criteria. For example, if we have a variable "maximum speed" for a class **Car**, it is possible, through the use of properties, to require its value to be within the range of 1 to 300 km/h.

## Physical Presentation of the Properties in a Class

Properties may have **different presentation in a class** at a physical level. As seen in our example, the information about the coordinates of the class **Point** was initially stored in two fields and later in a field-array.

If, instead of keeping the information about the coordinates of the point in a field, we decide to save it in a file or in a database, then every time we need to access the respective information, we can also read or write from the file or database rather than use the fields of the class as in the previous examples. Since properties are accessed by special methods (called methods for access and modification or **accessor methods**), to be discussed later, the question how information will be stored does not really matter to the classes that use our class, because of the encapsulation.

However, in general, information of the class is saved fields of the class, which have 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).**

## Property without Declaration of a Field

Consider an example for which a property is stored neither in the field, nor anywhere else, but calculated when accessed.

Let's consider the class **Rectangle**, which represents the geometric shape of a rectangle. Accordingly, this class has two fields – one for **width** and another for **height**. Assume that our class has one more property – **area**. Because we always can **calculate the property "area"** of a rectangle based on the width and the height, it is not required to define a separate field in the class to keep its value. Therefore, we can simply declare a method for calculating and obtaining the area of a rectangle:

### Rectangle.cs

```
using System;
```

```

class Rectangle
{
    private float height;
    private float width;

    public Rectangle(float height, float width)
    {
        this.height = height;
        this.width = width;
    }

    // Obtaining the value of the property area
    public float Area
    {
        get { return this.height * this.width; }
    }
}

```

As we will see, a property does not necessarily have both an accessing and a modifying method. It is allowed to declare only one method for reading the property **Area** of the rectangle. There is no point in having a method for modifying the value of the area of the rectangle, because the area is completely defined by given lengths of the sides.

## Declaring Properties in C#

To declare a property in C#, we have to declare access methods (for reading and writing) and decide how we will store the information accessed by the property in the class.

Before we declare methods we first have to declare the property. Formal declaration of properties occurs in the following way:

```
[<modifiers>] <property_type> <property_name>
```

With **<modifiers>** we denote both the **access modifiers and other modifiers** (e.g. **static**, to be discussed [in the next section of this chapter](#)). Modifiers are not a mandatory part of the declaration of a property.

The **type of the property <property\_type>** specifies the type of the variables of the property. They may be either primitive (e.g. **int**) or reference types (e.g. array).

The **<property\_name>** is **the name of the property**. It must begin with a capital letter and satisfy the **PascalCase** rule, i.e. every word in the property name starts with a capital letter. Here are some examples of properly named properties:

```

// MyValue property
public int MyValue { get; set; }

// Color property
public string Color { get; set; }

// X-coordinate property
public double X { get; set; }

```



## The Body of a Property

Like classes and methods in C#, properties also have **bodies**, where the methods for access are declared (**accessors**).

```
[<modifiers>] <property_type> <property_name>
{
    // ... Property's accessors methods go here
}
```

The body of a property begins with an opening curly bracket "{" and ends with a closing curly bracket – "}". Properties should always have a body.

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

As we explained, a **method for reading a value of a property** (called a **getter**) is declared in the body of a property by using the following syntax:

```
get { <accessor_body> }
```

The content of the code block surrounded by the angle brackets (<accessor\_body>) is similar to the contents of any other method. Execution of the code in the accessor body returns the result of the method.

The method for reading the value of a property must end with a **return** or **throw** keyword. The type of the return value has to be the same as the type of the property <property\_type> in the property declaration.

Calling a Method for Reading Property's Value

Let's consider another example of a property – **Age**, which is of type **int** and declared for a field in the same class:

```
private int age;                // Field declaration

public int Age                  // Property
declaration
{
    get { return this.age; }    // Getter declaration
}
```

Assume that the property **Age** from this example is declared in the class **Dog**. Then, the method for reading the value of the property is called with the "dot" notation, applied to a variable of the same type as the class in which the property is declared:

```
Dog dogInstance = new Dog();
// ...
int dogAge = dogInstance.Age;           // Getter
invocation
Console.WriteLine(dogInstance.Age);    // Getter invocation
```

The last two lines of the example show that when accessing the property name through the dot notation, then the getter method (method for reading the age value) is called automatically.

## Method for Modifying the Property Value (Setter)

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

The declaration in the body of the property has the following syntax:

```
set { <accessor_body> }
```

The content of the code block surrounded by angle brackets – **<accessor\_body>** is similar to the content of any other method. Execution of the code in the accessor body changes the value of the property. The method uses a hidden parameter, called **value**, which C# provides by default and that contains the new value of the property. The type of the parameter is the same as the type of the property.

Let's update the example of the property **Age** in the class **Dog**, to illustrate what we discussed so far:

```
private int age;                // Field declaration

public int Age                  // Property
declaration
{
    get { return this.age; }
    set { this.age = value; }    // Setter declaration
}
```

## Calling a Method for Modifying the Property's Value

The method for modifying the property's value is called via the "dot" notation, applied to the variable of the same type as the class in which the property is declared:

```
Dog dogInstance = new Dog();
// ...
dogInstance.Age = 3;                // Setter invocation
```

In the last line, the value 3 is assigned after the setter method of the property **Age** is called. In this way, the value is assigned to the setter method of the **Age** property and saved in **value**. The value of the variable **value** is then assigned to the field **age** from the class **Dog**.

## Assertion of the Input Values

It is best practice to **check the validity of the input value** before the setter method modifies the property and in case the input not valid, take necessary "measures". Generally, an exception is thrown in case of incorrect input data.

Consider again the example of the age of the dog. As we know, age has to be a positive number. To prevent someone from assigning a negative number to the property **Age**, we add the following validation at the beginning of the setter method:

```
public int Age
{
    get { return this.age; }
    set
    {
        // Take precaution: perform check for correctness
        if (value < 0)
        {
            throw new ArgumentException(
                "Invalid argument: Age should be a
positive number.");
        }
        // Assign the new correct value
        this.age = value;
    }
}
```

In case someone tries to assign a negative number to **Age**, the code will throw an exception of type **ArgumentException**, with details of the problem.

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 as to all other methods, which can change a field of the 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, which we will discuss in chapter "[High-Quality Programming Code](#)".

## Automatic Properties in C#

In C# we can define properties without explicitly defining the underlying field behind them. This is called defining **automatic properties**:

Point.cs
<pre>using System;  class Point {     public double X { get; set; }     public double Y { get; set; }      public Point(int x, int y)     {         this.X = x;         this.Y = y;     } }</pre>

```

class PointTest
{
    static void Main()
    {
        Point myPoint = new Point(2, 3);
        Console.WriteLine("The X coordinate is: " +
myPoint.X);
        Console.WriteLine("The Y coordinate is: " +
myPoint.Y);
    }
}

```

The above example declares a class **Point** with two **automatic properties**: **X** and **Y**. These properties do not have explicitly defined underlying fields and the compiler defines them during the compilation. It looks like the **get** and **set** methods are empty but in fact the compiler defines an underlying field and fills the body of the **get** and **set** accessors with some read / write code for the automatically defined underlying field.

**Use automatic properties for simple classes** if you want to write less code. However, keep in mind that when you use automatic properties, your control over the assigned values is limited. You might have difficulty in adding checks for invalid data.

## Types of Properties

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

1. **Read-only**, i.e. these properties have only a **get** method as shown by the example of the area of the rectangle.
2. **Write-only**, i.e. these properties have only a **set** method, but no method for reading the value of the property.
3. **Read-write**, the most common case, 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**. Developers have to decide whether someone should be allowed to change the value of a property, defining the property as read-only or as read / write. **Write-only** properties are used very rarely.

## Static Classes and Static Members

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.

We will first consider the **static elements** of a class, the fields, methods, properties and constructors, before the concept of the static class.

## For What Are the Static Elements Used?

Before we study the working of the static elements, let's have a look at the reasons for using them.

## Method to Sum Two Numbers

Let's imagine a class with a single method that always works in the same manner. For example, its task is to get two numbers and return their sum as a result. In this scenario, it doesn't matter exactly which object from that class is going to implement that method, since it will always do the same thing – adding two numbers together, independent from the calling object.

In practice, **the behavior of the method does not depend on the object state** (the values in the object fields). So why the need to create an object to implement that method, given that the method does not depend on any of the objects from that class? Why not just let the class implement that method?

## Instance Counter for Given Class

Consider a different scenario. Let's say we want to store in our program the current number of objects, which have been created by a certain class. How will we store that value, which represents **the number of created objects**?

As we know, we cannot define a variable as a class field, because for each created object there will be created a copy of that field, initialized with a default value. Every single object will store its own value for the number of objects and the objects will not be able to **share information**.

It looks like such a counter should be outside a class rather than inside it. In the following subsections we will find out how to deal with such a problem.

## What Is a Static Member?

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

Therefore, when we mark fields, methods or properties as **static**, we can use them without creating any object from the given class. All we need is access (visibility) to the class, so that we can call the object's static methods or its static fields and properties.



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

On the other hand, if we have created objects from a class, then its **static fields and properties will be shared**. There is only one copy of the static field or property, which is shared among all objects created from that class. Comparably, in the VB.NET language we have the **shared** instead of the **static** keyword.

## Static Fields

When we create objects from a class, then each holds different values in its fields. For example, consider again the class **Dog**:

```
public class Dog
{
    // Instance variables
    private string name;
```

```
    private int age;
}
```

There are two fields in the class, one for the name – **name** and another for the age – **age**. For every object, each of these fields has its own value, which is stored in a different place in the memory for each object.

However, if we need **shared fields** for all objects from a class. In order to achieve that, we have to use the **static** modifier in the field declarations. Such **static fields** 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 from a class **share** the static fields of that class.



**All objects created from a class (that are instances of that class) share the static fields of the class.**

## Declaration of Static Fields

Static fields are declared the same way as class fields. If there is also an access modifier, then the keyword **static** should be added after it.

```
[<access_modifier>] static <field_type> <field_name>
```

Here is how a field named **dogCount** would look like. The field stores information about the number of objects created from the class **Dog**:

Dog.cs
<pre>public class Dog {     // Static (class) variable     static int dogCount;      // Instance variables     private string name;     private int age; }</pre>

Static fields are created when we access them for the first time (read / modify). After their creation, they are initialized with the default values for their types.

## Initialization during Declaration

Static fields are initialized only once before they are accessed for the first time.

We can append the **static field initialization** to the above example:

```
// Static variable – declaration and initialization
static int dogCount = 0;
```

This initialization will complete during the first invocation of the static field. When we access a static field, memory will be reserved and the field will be initialized with a default value. If the field is initialized during its declaration (as in our case of the **dogCount** field), then initialization will execute immediately. If we try later to access the field from another part of our program, this initialization process will not repeat, because the static field exists and is already initialized.

## Accessing Static Fields

In contrast to common (non-static) class fields, static fields can be accessed directly by an external class. In order to do that we need to use the **"dot" notation** like this:

```
<class_name>.<static_field_name>
```

For example, if we want to print the value of the static field that holds the number of objects created from our class **Dog**, then we should do it like this:

```
static void Main()
{
    // Access to the static variable through class name
    Console.WriteLine("Dog count is now " + Dog.dogCount);
}
```

The result of the **Main()** method is:

```
Dog count is now 0
```

If we have a field in the class, which is defined as static, then we can access it directly without using the class name, because it is known by default, shared and unique.

```
<static_field_name>
```

## Modification of the Static Field Values

As mentioned before, static variables are **shared between all objects** from the class and do not belong to any particular object. Therefore, each object can access and modify the value of a shared static field, while simultaneously, all other objects from the same class can "see" the modified value.

That's why, if we want to count the number of objects created from the class **Dog**, we should use a **static field** to store the number and increment it by one every time the constructor is invoked, i.e. every time we create an object from our class.

```
public Dog(string name, int age)
{
    this.name = name;
    this.age = age;

    // Modifying the static counter in the constructor
    Dog.dogCount += 1;
}
```

We access a static field from the class **Dog** so we can use the following code in order to access the static field **dogCount**:

```
public Dog(string name, int age)
{
    this.name = name;
    this.age = age;

    // Modifying the static counter in the constructor
    dogCount += 1;
}
```

The first way is preferable, as it is clearer that the field in the class **Dog** is static. The code is more readable.

Let's create some objects from the class **Dog** and print their number for control:

```
static void Main()
{
    Dog dog1 = new Dog("Jackie", 1);
    Dog dog2 = new Dog("Lassy", 2);
    Dog dog3 = new Dog("Rex", 3);

    // Access to the static variable
    Console.WriteLine("Dog count is now " + Dog.dogCount);
}
```

The output of the example is:

```
Dog count is now 3
```

## Constants

Before we finish with the static fields, we should get familiar with one more specific type of static field.

Like the constants of mathematics, C# allows creation of special fields of a class called **constants**. Once declared and initialized, **constants always retain the same value** for all objects of a particular type.

In C#, constants are of two types:

1. Values which are declared during the compilation of the program (**compile-time constants**).
2. Values which are declared during the execution of the program (**run-time constants**).

### Compile-Time Constants (const)

Constants which are calculated at compile time (compile-time constants) are declared as follows, using the modifier **const**:

```
[<access_modifiers>] const <type> <name>;
```



Constants which are declared with the special word **const** are static fields. However, the use of the modifier **static** is not allowed.



**Although the constants declared with a modifier `const` are static fields, cannot have the `static` modifier in their declaration.**

For example, we declare the constant number "PI", known from mathematics, as follows:

```
public const double PI = 3.141592653589793;
```

The value we assign to a constant can also be an expression, which has to be calculated by the compiler at compile time. For example, as we may know from mathematics, the constant "PI" can also be represented by the approximate result of the division of the numbers 22 and 7:

```
public const double PI = 22d / 7;
```

When we print the value of this constant:

```
static void Main()
{
    Console.WriteLine("The value of PI is: " + PI);
}
```

Then the command line will display:

```
The value of PI is: 3.14285714285714
```

If we do not provide a value to the **const** constant at its declaration, but try to do it later, then we will get a compilation error. For example, if we first declare the constant PI and then later try to give it a value:

```
public const double PI;

// ... Some code ...

public void SetPiValue()
{
    // Attempting to initialize the constant PI
    PI = 3.141592653589793;
}
```

Then the compiler will issue an error like the one below:

```
A const field requires a value to be provided
```

Let's pay attention:



**Constants declared with modifier `const` must be initialized at the moment of their declaration.**

## Assigning Constant Values at Runtime

Having learned how to declare constants that are being initialized at compile time, let's now consider another example. Suppose, we want to create a class for color (**Color**). We will use the so-called **Red-Green-Blue (RGB) color model**, according to which each color can be made from a mixture of the three primary colors – red, green and blue. These three primary colors are represented by three integers in the range from 0 to 255. For example, black is represented as (0, 0, 0), white as (255, 255, 255), blue as (0, 0, 255) etc.

In our class, we declare three integer fields for red, green and blue. As well as a constructor that accepts values for each of them:

Color.cs
<pre>class Color {     private int red;     private int green;     private int blue;      public Color(int red, int green, int blue)     {         this.red = red;         this.green = green;         this.blue = blue;     } }</pre>

As some colors are used more frequently than others (for example black and white) we can **declare constants for them**, with the idea that the users of our class will take them for granted, instead of creating every time their own objects for these particular colors. To do this, we modify the code of our class by adding the declaration of the following color-constants:

Color.cs
<pre>class Color {     public const Color Black = new Color(0, 0, 0);     public const Color White = new Color(255, 255, 255);      private int red;     private int green;     private int blue;      public Color(int red, int green, int blue)     {         this.red = red;         this.green = green;         this.blue = blue;     } }</pre>

```
}
```

Strangely, when we try to compile we **get the following error**:

```
'Color.Black' is of type 'Color'. A const field of a reference
type other than string can only be initialized with null.
'Color.White' is of type 'Color'. A const field of a reference
type other than string can only be initialized with null.
```

This is so because in C#, constants declared with the modifier **const** can only be of the following types:

1. Primitive types: **sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool**.
2. **Enumerated types** (discussed in section "[Enumerations](#)" at the end of this chapter).
3. **Reference types** (mostly the type **string**).

The problem with the compilation of the class in our example is connected to the reference type and the restriction by the compiler forbidding the use of the operator **new** when declaring a constant, if the constant has already been declared with the modifier **const**, unless this constant is a reference type that can be calculated at compile time.

As we might 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 a reference type other than **string**, is **null**.

We can formulate the following definition:



**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`.**

Thus, using the modifier **const**, we will not be able to declare the constants **Black** and **White** of type **Color** in our color class, because they aren't **null**. The next section will show how to deal with this problem.

## Runtime Constants (readonly)

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 a **const** modifier.

```
[<access_modifiers>] static readonly <reference-type> <name>;
```

The value of the type **<reference-type>** cannot be defined at compilation time.

If we replace **const** by **static readonly** in the last example of the previous section, then the code will compile successfully:

```
public static readonly Color Black = new Color(0, 0, 0);  
public static readonly Color White = new Color(255, 255, 255);
```

## Naming the Constants

The naming of constants in C# follow the **PascalCase** rule in line with Microsoft's official C# coding convention. If the constant is composed of several words, then each word begins with a capital letter. Here are some examples of correctly named constants:

```
// The base of the natural logarithms (approximate value)  
public const double E = 2.718281828459045;  
public const double PI = 3.141592653589793;  
public const char PathSeparator = '/';  
public const string BigCoffee = "big coffee";  
public const int MaxValue = 2147483647;  
public static readonly Color DeepSkyBlue = new Color(0,104,139);
```

Naming in the style of **ALL-CAPS** is not officially supported by the Microsoft code convention, although it is widely distributed in programming:

```
public const double FONT_SIZE_IN_POINTS = 14; // 14pt font size
```

The examples show that the difference between **const** and **static readonly** fields is in the moment of their value assignments. The compile-time 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.

## Using Constants

Constants are used in programming to **avoid repetition of numbers, strings or other shared values** (literals). The use of constants, instead of brutally hardcoded, repeating values, improves readability and facilitates maintenance of the code and, therefore, is a highly recommended practice. According to some authors, all literals other than **0**, **1**, **-1**, empty string, **true**, **false** and **null** must be declared as constants, but this complicates the reading and maintenance of the code, instead of simplifying it. Therefore, it is advised that **values, which occur more than once in the program or are likely to change over time, are declared as constants.**

In the chapter "[High-Quality Programming Code](#)", we will learn in detail when and how to use constants efficiently.

## Static Methods

Like static fields, we declare methods static if we want to associate them only with a class and not with the object from the class.

### Declaration of Static Methods

To declare a **static method**, we add the keyword **static** to the method's declaration:

```
[<access_modifier>] static <return_type> <method_name>()
```

Let's for example declare as static the method for summing two numbers, which we discussed at the beginning of this section:

```
public static int Add(int number1, int number2)
{
    return (number1 + number2);
}
```

## Accessing Static Methods

Like static fields, static methods can be **accessed with the "dot" notation** (the dot operator) preceded by the name of the class. The class name can be omitted if the method is called from its own class. Here is an example of calling the static method **Add(...)**:

```
static void Main()
{
    // Call the static method through its class
    int sum = MyMathClass.Add(3, 5);

    Console.WriteLine(sum);
}
```

## Access between Static and Non-Static Members

In most cases, **static methods are used to access static fields** in the class where they have been defined. For example, if we want to declare a method, which returns the number of created objects of the **Dog** class, it must be static, because our counter will be static too:

```
public static int GetDogCount()
{
    return dogCount;
}
```

When we examine how static and non-static methods and fields can be accessed, it appears that not all combinations are allowed.

## Accessing Non-Static Members from Non-Static Method

Non-static methods can access non-static fields and other non-static methods of the class. For example, in the **Dog** class, we can declare the method **PrintInfo()**, which displays information about our dog:

Dog.cs
<pre>public class Dog {     // Static variable     static int dogCount;</pre>

```

// Instance variables
private string name;
private int age;

public Dog(string name, int age)
{
    this.name = name;
    this.age = age;

    dogCount += 1;
}

public void Bark()
{
    Console.WriteLine("wow-wow");
}

// Non-static (instance) method
public void PrintInfo()
{
    // Accessing instance variables - name and age
    Console.WriteLine("Dog's name: " + this.name + "; age: "
        + this.age + "; often says: ");

    // Calling instance method
    this.Bark();
}
}

```

If we create an object from the **Dog** class and call its **PrintInfo()** method:

```

static void Main()
{
    Dog dog = new Dog("Doggy", 1);
    dog.PrintInfo();
}

```

The result will be the following:

```
Dog's name: Doggy; age: 1; often says: wow-wow
```

## Accessing Static Elements from Non-Static Method

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

For example:

### Circle.cs

```
public class Circle
{
    public static double PI = 3.141592653589793;

    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public static double CalculateSurface(double radius)
    {
        return (PI * radius * radius);
    }

    public void PrintSurface()
    {
        double surface = CalculateSurface(radius);
        Console.WriteLine("Circle's surface is: " +
surface);
    }
}
```

In this example, we provide access to the value of the static field **PI** from the non-static method **PrintSurface()**, by calling the static method **CalculateSurface()**. Let's call the non-static method:

```
static void Main()
{
    Circle circle = new Circle(3);
    circle.PrintSurface();
}
```

After the compilation and the execution, the following result will be printed on the console:

```
Circle's surface is: 28.2743338823081
```

### Accessing Static Elements of the Class from Static Method

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

For example, let's consider our class for mathematical calculations, where we declared the constant **PI**. We can declare a static method for finding the circumference (perimeter) of the circle (the formula is  $2\pi r$ , where **r** is the radius of the circle). This method uses the constant **PI**. To demonstrate that a static method can call another static method, we call the static method for finding the circumference from the static method **Main()**:

#### MyMathClass.cs

```
public class MyMathClass
{
    public const double PI = 3.141592653589793;

    // The method applies the formula: P = 2 * PI * r
    static double CalculateCirclePerimeter(double r)
    {
        // Accessing the static variable PI from static
method
        return (2 * PI * r);
    }

    static void Main()
    {
        double radius = 5;

        // Accessing static method from other static method
        double circlePerimeter =
CalculateCirclePerimeter(radius);

        Console.WriteLine("Circle with radius " + radius +
            " has perimeter: " + circlePerimeter);
    }
}
```

The code is compiled without errors and displays the following output:

```
Circle with radius 5.0 has perimeter: 31.4159265358979
```

### Accessing Non-Static Elements from Static Method

Let's have a look at a most interesting case of **accessing non-static elements from a static method**.

From a static method we can neither access non-static fields, nor call non-static methods. This is because static methods are bounded by the class and do not "know" any object created from the class. Therefore, the keyword **this** cannot be used in static methods – it refers to a specific instance of the class. When we try to access non-static elements of the class (fields or methods) from a static method, we will always get a compilation error.

### Unauthorized Access to Non-Static Field – Example

If in our class **Dog** we declare a static method **PrintName()**, which has to return as result the value of the non-static field **name**:

```
public static void PrintName()
{
    // Trying to access non-static variable from static method
    Console.WriteLine(name); // INVALID
}
```



```
}
```

The compiler will respond with an **error message**:

```
An object reference is required for the non-static field, method,
or property 'Dog.name'
```

If we try to access the field in the method via the **keyword this**:

```
public void string PrintName()
{
    // Trying to access non-static variable from static method
    Console.WriteLine(this.name); // INVALID
}
```

The compiler again **fails to compile** the class and displays the following message:

```
Keyword 'this' is not valid in a static property, static method,
or static field initializer
```

### Illegal Call of Non-Static Method from Static Method – Example

Now we will try to call a non-static method from a static method. Let's first declare in our class **Dog**, the non-static method **PrintAge()**, which prints the value of the field **age**:

```
public void PrintAge()
{
    Console.WriteLine(this.age);
}
```

Then we try to call this non-static method from the static method **Main()**, declared in the class **Dog**, without creating an object from our class:

```
static void Main()
{
    // Attempt to invoke non-static method from a static
    context
    PrintAge(); // INVALID
}
```

When compiling, we will **get the following error**:

```
An object reference is required for the non-static field, method,
or property 'Dog.PrintAge()'
```

The result is similar, if we try to fool the compiler by calling the method via the keyword **this**:

```
static void Main()
{
```

```
// Attempt to invoke non-static method from a static
context
this.PrintAge(); // INVALID
}
```

Again, the compiler displays an error message and **fails to compile our class**:

Keyword 'this' is not valid in a static property, static method, or static field initializer

From the examples, we can draw the following conclusion:



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

The problem with the access to non-static elements of a class from a static method has a solution – non-static elements must be accessed by reference to an object:

```
static void Main()
{
    Dog myDog = new Dog("Lassie", 2);
    string myDogName = myDog.name;
    Console.WriteLine("My dog \" + myDogName + "\" has age of
");
    myDog.PrintAge();
    Console.WriteLine("years");
}
```

When this code is compiled, the result is:

My dog "Lassie" has age of 2 years

## Static Properties of the Class

In rare cases, it is convenient to declare and use the characteristics of the class, rather than of the object. Static properties have the same characteristics as properties of an object from a class, but **static properties refer to the class** (instead of its objects).

All we need to do, to turn a property into a static one, is to **add the static keyword in its declaration**.

The static properties are **declared** as follows:

```
[<modifiers>] static <property_type> <property_name>
{
    // ... Property's accessors methods go here
}
```

Let's consider an example. We have a class that describes a system and that has many objects created from it. In order to ensure that version and vendor properties of the system are the same for all instances created from this class, we make the properties of the class static.:

### SystemInfo.cs

```
public class SystemInfo
{
    private static double version = 0.1;
    private static string vendor = "Microsoft";

    // The "version" static property
    public static double Version
    {
        get { return version; }
        set { version = value; }
    }

    // The "vendor" static property
    public static string Vendor
    {
        get { return vendor; }
        set { vendor = value; }
    }

    // ... More (non)static code here ...
}
```

In this example, we store the values of the static properties in static fields (which is logical, since they belong only to the class). The static properties **Version** and **Vendor** are accessible by the objects from this class. If someday, the system version is upgraded and, for instance, becomes version **0.2**, then each object receives the new version when accessing the class version property.

### Static Properties and the Keyword "this"

Like static methods, the keyword **this** cannot be used in the static properties, as the static property is associated only with the class and does not "recognize" objects from a class.



**The keyword `this` cannot be used in static properties.**

### Accessing Static Properties

Like the static fields and methods, static properties can be accessed by "**dot**" notation, applied only to the name of the class in which they are declared.

To be sure, let's try to access the property **Version** through a variable of the class **SystemInfo**:

```
static void Main()
{
    SystemInfo sysInfoInstance = new SystemInfo();
    Console.WriteLine("System version: " +
        sysInfoInstance.Version);
}
```

```
}
```

When we try to compile the above code, we get the following error message:

```
Member 'SystemInfo.Version.get' cannot be accessed with an  
instance reference; qualify it with a type name instead
```

However, if we try to access the static properties through the class name, the code compiles and works correctly:

```
static void Main()  
{  
    // Invocation of static property setter  
    SystemInfo.Vendor = "Microsoft Corporation";  
  
    // Invocation of static property getters  
    Console.WriteLine("System version: " +  
        SystemInfo.Version);  
    Console.WriteLine("System vendor: " + SystemInfo.Vendor);  
}
```

After the code is compiled, the result of its execution is:

```
System version: 0.1  
System vendor: Microsoft Corporation
```

Before proceeding to the next section, let's look at the printed value of the property **Vendor**. It is "Microsoft Corporation", although we have initialized it with the value "Microsoft" in the **SystemInfo** class. This is because we changed the value of the property **Vendor** in the first line of the **Main()** method, by calling its method for modification.



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

## Static Classes

For a complete understanding, we have to explain that one can also declare classes as static. Similar to static members, a class is static when the keyword **static** is used in its declaration.

```
[<modifiers>] static class <class_name>  
{  
    // ... Class body goes here  
}
```

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

Static classes are rarely used and then essentially associated with the **use of static methods and constants**, which do not belong to any particular object. The details of static classes go beyond the scope of this book. Curious readers can find more information on the site of the

## Static Constructors

To complete the section on static class members, we should mention that classes may also have a **static constructor** (i.e. a constructor that has the **static** keyword in its declaration):

```
[<modifiers>] static <class_name>([<parameters_list>])  
{  
}
```

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

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

### Static Constructor – Example

Consider an example for the **use of a static constructor**. We want to make a class that quickly calculates the square root of an integer and returns the whole part of the result, which is also an integer. Since calculating the square root is a time-consuming mathematical operation, involving calculations with real numbers and calculating convergent series, it is a good idea to do these calculations only once, at program startup, and then re-use the already calculated values. Of course, to **pre-compute all square roots** within a given range, we must first define the range, which should not be too wide (e.g. from 1 to 1000). Then, at the first request for a square root, we have to calculate all the square roots within the range and return the calculated values. Upon a following request for a square root, all values are available and can be returned immediately. If the program is never required to calculate a square root, then preliminary calculation should not occur at all.

Initially, CPU time is invested in preliminary calculation, thereafter extraction of the square roots is very fast. If we need multiple calculations of a square root, then pre-calculation will significantly increase performance.

All this can be implemented in a **static class with a static constructor** that calculates square roots. The calculations are **stored in a static array**. A **static method** is used to extract the pre-calculated values. Since the code for initial calculations is in a static constructor, CPU time and memory will be saved, as long as the class is not used and no calculation is performed.

This is how the implementation might look like:

```
static class SqrtPrecalculated  
{  
    public const int MaxValue = 1000;  
  
    // Static field  
    private static int[] sqrtValues;  
  
    // Static constructor
```

```

static SqrtPrecalculated()
{
    sqrtValues = new int[MaxValue + 1];
    for (int i = 0; i < sqrtValues.Length; i++)
    {
        sqrtValues[i] = (int)Math.Sqrt(i);
    }
}

// Static method
public static int GetSqrt(int value)
{
    if ((value < 0) || (value > MaxValue))
    {
        throw new
ArgumentOutOfRangeException(String.Format(
    "The argument should be in range
[0...{0}].",
        MaxValue));
    }
    return sqrtValues[value];
}

class SqrtTest
{
    static void Main()
    {
        Console.WriteLine(SqrtPrecalculated.GetSqrt(254));
        // Result: 15
    }
}

```

## Structures

In C# and .NET Framework, there are two implementations of the concept of "class" for object-oriented programming: **classes** and **structures**. Classes are defined through the keyword **class**, while the structures are defined through the keyword **struct**. 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).

## Structure (struct) – Example

Let's define a **structure** to hold a point in the 2D space, similar to the class **Point** defined in the section "[Example of Encapsulation](#)":

#### Point2D.cs

```
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; }
    }
}
```

The only difference is that now we defined **Point2D** as **struct**, not as **class**. **Point2D** is a structure, a value type, so its instances behave like **int** and **double**. They are value types (not objects), which means they cannot be **null** and as method parameters, they are **passed by value**.

## Structures are Value Types

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

```
class PlayWithPoints
{
    static void PrintPoint(Point2D p)
    {
        Console.WriteLine("{0},{1}", p.X, p.Y);
    }

    static void TryToChangePoint(Point2D p)
    {
        p.X++;
        p.Y++;
    }

    static void Main()
    {
    }
}
```

```

{
    Point2D point = new Point2D(3, -2);
    PrintPoint(point);
    TryToChangePoint(point);
    PrintPoint(point);
}

```

If we run the above example, the result will be as follows:

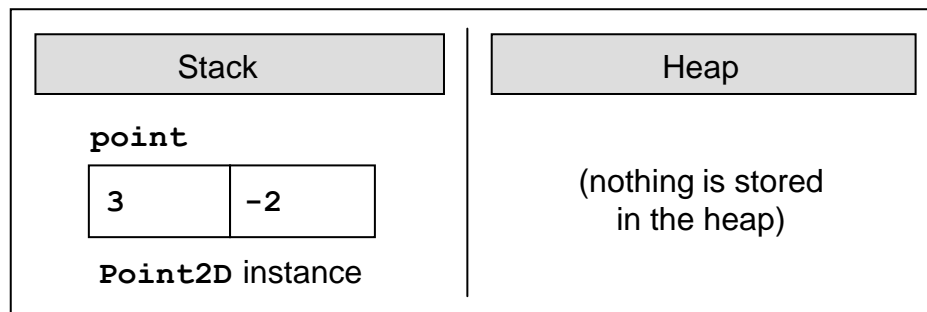
```

(3, -2)
(3, -2)

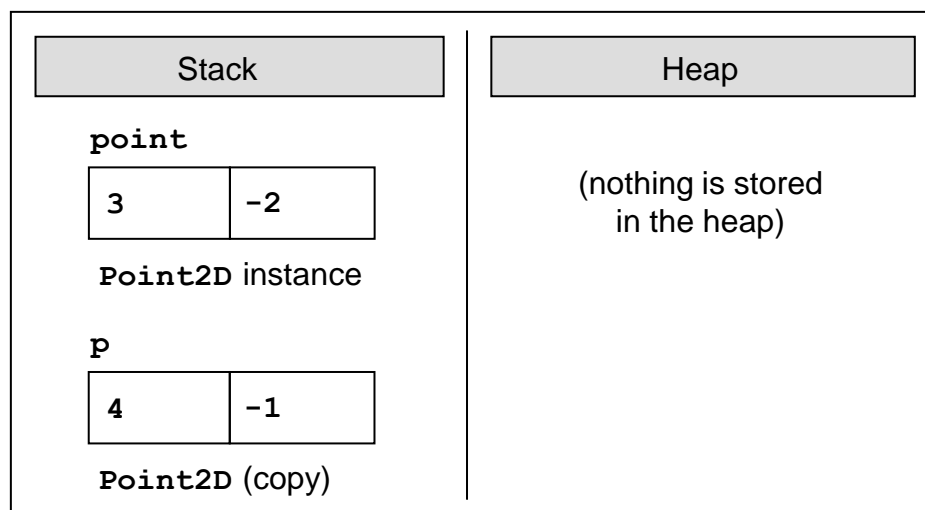
```

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.

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

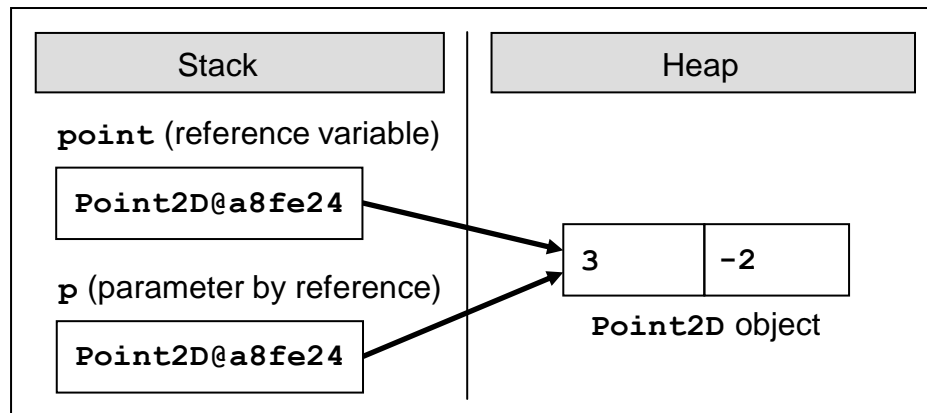




If we change **Point2D** from **struct** to **class**, the result will be very different:

```
(3, -2)
(4, -1)
```

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:



## Class or Structure?

How to decide **when to use a class and when a structure**? We will give you some general guidelines.

**Use structures** to hold simple data structures consisting of few fields that belong together. Examples are coordinates, sizes, locations, colors, etc. Structures are not supposed to have functionality inside (no methods except simple ones like **ToString()** and comparators). Use structures for **small data structures consisting of set of fields** that should be passed by value.

**Use classes** for more complex scenarios where you combine data and programming logic into a class. If you have logic, use a class. If you have more than a few simple fields, use a class. If you need to pass variables by reference, use a class. If you need to assign a **null** value, preferably use a class. If you prefer working with a reference type, use a class.

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

There are few other **differences between class and structure** in addition to classes being reference types and structures values types, but we will not further discuss them. For more detail, we refer to the following article in MSDN: <http://msdn.microsoft.com/en-us/library/ms173109.aspx>.

## Enumerations

[Earlier in this chapter](#), we discussed what constants are, and how to declare and use them. In this context, we will now consider a part of the C# language, in which a variety of logically connected constants can be linked by means of language. These language constructs are the so-called **enumerated types**.

## Declaration of Enumerations

An **Enumeration** is a structure, which resembles a class but differs from it, in that in its body we can **declare only constants**. Enumerations can take values only from the constants listed in the type. An enumerated variable can have one of the listed type constant values but cannot have the value **null**.

Enumerations are declared using the reserved word **enum** instead of **class**:

```
[<modifiers>] enum <enum_name>
{
    constant1 [, constant2 [, [, ... [, constantN]]
}
```

The **<modifiers>** are the access modifiers **public**, **internal** and **private**. The identifier **<enum\_name>** follows the rules for class names in C#. Constants separated by commas are declared in the enumeration code block.

Consider an example. Let's define an enumeration for the days of the week (we will call it **Days**). The constants that will appear in this particular enumeration are the **names of the days of the week**:

Days.cs
<pre>enum Days {     Mon, Tue, Wed, Thu, Fri, Sat, Sun }</pre>

Naming of constants in one particular enumeration follows the same principles as explained in the "[Naming Constants](#)" section.

Each of the constants listed in the enumeration is of the type of the enum, i.e. in our case **Mon** belongs to type **Days**.

If we execute the following line:

```
Console.WriteLine(Days.Mon is Days);
```

This will be printed as a result:

```
True
```

Let's repeat:



**An enumeration is a list of constants of the type – of the enum.**

## Nature of Enumerations

Each constant, which is declared in an enumeration, is associated with an underlying integer type, by default **int**.

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

```
int mondayValue = (int)Days.Mon;  
Console.WriteLine(mondayValue);
```

After we execute it, the result will be:

```
0
```

The values associated with constants of a particular enumerated type by default are the indices of the list of constants of this type, i.e. numbers from 0 to the number of constants in the type minus 1. The constant **Mon** is associated with the integer value 0, the constant **Tue** with 1, **Wed** – with 2, etc.



**Each constant in an enumeration is actually a textual representation of an integer. By default, this number is the index of the list of constants of the enumeration type.**

Despite the integer index of the constants of an enumeration, their declared textual representation will be printed:

```
Console.WriteLine(Days.Mon);
```

Prints the following result:

```
Mon
```

## Hidden Numerical Value of Constants in Enumeration

It is possible to change the **numerical value of constants in an enumeration** during their declaration:

```
[<modifiers>] enum <enum_name>  
{  
    constant1[=value1] [, constant2[=value2] [, ... ]]  
}
```

Where **value1**, **value2**, etc. must be integers.

For better understanding, consider a class **Coffee**, which represents a cup of coffee that customers order in a coffee shop:

### Coffee.cs

```
public class Coffee  
{  
    public Coffee()  
    {  
    }  
}
```

```
}
```

Customers can order different amounts of coffee. The coffee machine has the predefined values "small" – 100 ml, "normal" – 150 ml and "double" – 300 ml. Therefore, we declare an enumeration **CoffeeSize**, which has three constants – **Small**, **Normal** and **Double**, to which the respective quantities are assigned:

#### CoffeeSize.cs

```
public enum CoffeeSize
{
    Small=100, Normal=150, Double=300
}
```

Now we can add a field and property to the class **Coffee**, which store the type of coffee the customer has ordered:

#### Coffee.cs

```
public class Coffee
{
    public CoffeeSize size;

    public Coffee(CoffeeSize size)
    {
        this.size = size;
    }

    public CoffeeSize Size
    {
        get { return size; }
    }
}
```

Let's print the values of the coffee quantity for a normal and double coffee:

```
static void Main()
{
    Coffee normalCoffee = new Coffee(CoffeeSize.Normal);
    Coffee doubleCoffee = new Coffee(CoffeeSize.Double);

    Console.WriteLine("The {0} coffee is {1} ml.",
        normalCoffee.Size, (int)normalCoffee.Size);
    Console.WriteLine("The {0} coffee is {1} ml.",
        doubleCoffee.Size, (int)doubleCoffee.Size);
}
```

After compiling and executing, the following is printed:

```
The Normal coffee is 150 ml.
```

The Double coffee is 300 ml.

## Use of Enumerations

The main purpose of the enumerations is to simplify code and make it easier to understand by **replacing numeric values with text strings**.

Another very important advantage of enumerations is the requirement by the compiler to use the constants of the enumeration instead of numbers, minimizing the chance on errors in the code. For example, if we use an `int` variable instead of a constant from the enumeration, nothing prevents us from later assigning a wrong value to that integer variable.

To clarify this, consider the following example: a class "**coffee price calculator**", which calculates the price of each type of coffee offered in the coffee shop:

```
PriceCalculator.cs

public class PriceCalculator
{
    public const int SmallCoffeeQuantity = 100;
    public const int NormalCoffeeQuantity = 150;
    public const int DoubleCoffeeQuantity = 300;

    public CashMachine() { }

    public double CalcPrice(int quantity)
    {
        switch (quantity)
        {
            case SmallCoffeeQuantity:
                return 0.20;
            case NormalCoffeeQuantity:
                return 0.30;
            case DoubleCoffeeQuantity:
                return 0.60;
            default:
                throw new InvalidOperationException(
                    "Unsupported coffee quantity: " +
quantity);
        }
    }
}
```

We have three constants for the different capacities of the coffee cups in the coffee shop, respectively, 100, 150 and 300 ml. Furthermore, **we expect** that users of our class will diligently use the defined constants, instead of numbers – **SmallCoffeeQuantity**, **NormalCoffeeQuantity** and **DoubleCoffeeQuantity**. The method **CalcPrice(int)** returns the respective price, calculating it from the submitted amount.

The problem lies in the fact that someone may decide not to use the constants defined by us, but submit an invalid number as a parameter for our method. For example: -1 or 101. In that

case, without the current checking for invalid quantities, the method would return a wrong price and behave incorrectly.

To avoid this problem, we use a particular feature of these enumerations, namely that constants in the enumeration type can be used in a **switch-case** structure. They can be submitted as values of the operator **switch** and accordingly – as operands of the operator **case**.



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

Let's rework the method which calculates the price for a cup of coffee depending on the capacity of the cup. This time we will use the enumeration type **CoffeeSize**, which we declared previously:

```
public double CalcPrice(CoffeeSize coffeeSize)
{
    switch (coffeeSize)
    {
        case CoffeeSize.Small:
            return 0.20;
        case CoffeeSize.Normal:
            return 0.40;
        case CoffeeSize.Double:
            return 0.60;
        default:
            throw new InvalidOperationException(
                "Unsupported coffee quantity: " +
                (int)coffeeSize);
    }
}
```

The users of our method can no longer cause unexpected behavior, because they have to use specific arguments, namely the constants of the enumerated **CoffeeSize** type. This is one of the advantages of constants which are declared in enumeration types, as compared to constants declared in a class.



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

Before completing the enumeration section, we remark that enumerations should be used with caution when working with the **switch-case** statement. For example, if one day the owner of the coffee shop wants to serve mugsof coffee, then a new constant has to be added to the list of the enumeration **CoffeeSize**, which may be called, for example, **Overwhelming**:

#### CoffeeSize.cs

```
public enum CoffeeSize
{
    Small=100, Normal=150, Double=300, Overwhelming=600
}
```

When calculating the coffee price of the new quantity, the method which calculates the price will throw an exception, informing the user about a case of "Unsupported coffee quantity".

What we should do to solve this problem is to add a new **case**-condition, which reflects the new constant in the enumerated **CoffeeSize** type.



**When we modify the list of constants in an existing enumeration, we should be careful not to break the logic of the existing code that uses the declared constants.**

## Inner Classes (Nested Classes)

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

The main reasons to declare one class into another are:

1. To **better organize the code** when working with objects in the real world, which have a special relationship and when one cannot exist without the other.
2. To **hide a class in another class**, so that the inner class cannot be used from outside the outer class wrapping it.

In general, inner classes are used rarely, because they complicate the structure of the code as levels of nesting increase.

## Declaration of Inner Classes

Inner classes are declared in the same way as normal classes, but are **located within another class**. Allowed modifiers in the declaration of the inner class are:

1. **public** – an inner class is accessible from any assembly.
2. **internal** – an inner class is accessible only from the assembly of the outer class.
3. **private** – access is restricted to the outer class wrapping the inner class.
4. **static** – the 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**, used in an inner class, only refers to the internal class. Fields of the outside class **cannot be accessed** using the reference **this**. If fields of the outer class need to be accessed from the internal class A reference to the outer class needs to be passed as a parameter to the inner class.

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

## Inner Classes – Example

Consider the following example:

OuterClass.cs
<pre>public class OuterClass</pre>

```

{
    private string name;

    private OuterClass(string name)
    {
        this.name = name;
    }

    private class NestedClass
    {
        private string name;
        private OuterClass parent;

        public NestedClass(OuterClass parent, string name)
        {
            this.parent = parent;
            this.name = name;
        }

        public void PrintNames()
        {
            Console.WriteLine("Nested name: " +
this.name);
            Console.WriteLine("Outer name: " +
this.parent.name);
        }

        static void Main()
        {
            OuterClass outerClass = new OuterClass("outer");
            NestedClass nestedClass = new
                OuterClass.NestedClass(outerClass, "nested");
            nestedClass.PrintNames();
        }
    }
}

```

In the example, the outer class **OuterClass** contains the inner class **InnerClass**. By using the keyword **this**, the non-static inner class methods can access inner class members, as well as an instance of the outer class **parent** (through **this.parent**, if the **parent** reference is added as a parameter).

If we run the above example, we will obtain the following result:

```

Nested name: nested
Outer name: outer

```

## Usage of Inner Classes

Let's have a class for a car – **Car**. Each car has an engine and doors. Unlike the car's doors, the engine is an intrinsic part of the inner workings of a car and we are dealing with a case of



so-called composition (for further explanation see the section "[Class Diagrams: Composition](#)" in the chapter "[Principles of Object-Oriented Programming](#)").



**When the connection between two classes is a composition, then it is convenient to declare an inner class within an outer class.**

Therefore, if you declare the class **Car**, then it is appropriate to create its inner class **Engine**, reflecting the concept of a car engine inside a car:

Car.cs
<pre>class Car {     Door FrontRightDoor;     Door FrontLeftDoor;     Door RearRightDoor;     Door RearLeftDoor;     Engine engine;      public Car()     {         engine = new Engine();         engine.horsePower = 2000;     }      public class Engine     {         public int horsePower;     } }</pre>

## Declare Enumeration in a Class

In some cases, **enumeration is declared within a class** for better encapsulation of the class.

For example, the enumeration of type **CoffeeSize**, which we created in the [previous section](#), can be declared inside the class **Coffee**, thereby improving its encapsulation:

Coffee.cs
<pre>class Coffee {     // Enumeration declared inside a class     public static enum CoffeeSize     {         Small = 100, Normal = 150, Double = 300     }      // Instance variable of enumerated type</pre>

```

        private CoffeeSize size;

        public Coffee(CoffeeSize size)
        {
            this.size = size;
        }

        public CoffeeSize Size
        {
            get { return size; }
        }
    }

```

Consequently, the method for calculation of the price of coffee has to be slightly modified:

```

public double CalcPrice(Coffee.CoffeeSize coffeeSize)
{
    switch (coffeeSize)
    {
        case Coffee.CoffeeSize.Small:
            return 0.20;
        case Coffee.CoffeeSize.Normal:
            return 0.40;
        case Coffee.CoffeeSize.Double:
            return 0.60;
        default:
            throw new InvalidOperationException(
                "Unsupported coffee quantity: " +
                ((int)coffeeSize));
    }
}

```

## Generics

In this section we will explain the concept of **generic classes** (generic data types, generics). First, let's examine an example that helps us understand the idea.

### Shelter for Homeless Animals – Example

Let's assume that we have two classes. A class **Dog**, which describes a dog:

Dog.cs
<pre> public class Dog { } </pre>

And a class **Cat**, which describes a cat:

Cat.cs
--------

```
public class Cat
{
}
```

We like to create a class that describes a **shelter for homeless animals – AnimalShelter**. This shelter has a specific number of free cells, which determines the number of animals that can find refuge in the shelter. A special feature of the shelter class has to guarantee separate accommodation of animals of different kind, in our case, dogs or cats, because putting together different species might not be such a good idea.

If we want to solve the problem with what we have learned so far,, then we might come up with a following conclusion – to ensure that our class will contain elements only from one and the same type we need to use an array of identical objects. These objects may be dogs, cats or simply instances of a universal type **object**.

For instance, if we want to make a shelter for dogs, here is how our class might look like:

```
AnimalsShelter.cs

public class AnimalShelter
{
    private const int DefaultPlacesCount = 20;

    private Dog[] animalList;
    private int usedPlaces;

    public AnimalShelter() : this(DefaultPlacesCount)
    {
    }

    public AnimalShelter(int placesCount)
    {
        this.animalList = new Dog[placesCount];
        this.usedPlaces = 0;
    }

    public void Shelter(Dog newAnimal)
    {
        if (this.usedPlaces >= this.animalList.Length)
        {
            throw new InvalidOperationException("Shelter
is full.");
        }
        this.animalList[this.usedPlaces] = newAnimal;
        this.usedPlaces++;
    }

    public Dog Release(int index)
    {
        if (index < 0 || index >= this.usedPlaces)
        {
        }
    }
}
```

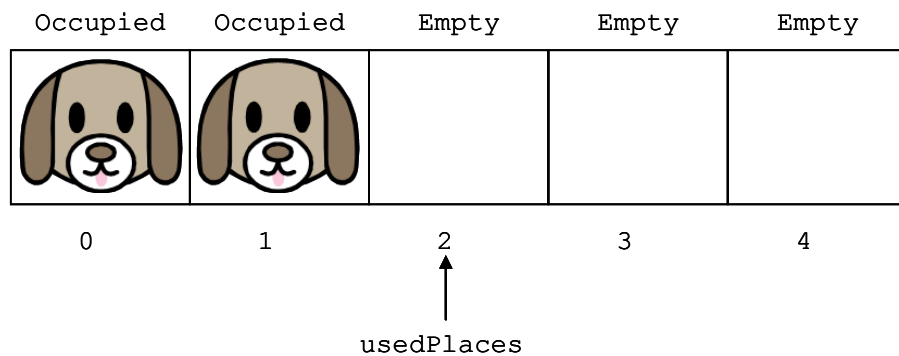
```

        throw new ArgumentOutOfRangeException(
            "Invalid cell index: " + index);
    }
    Dog releasedAnimal = this.animalList[index];
    for (int i = index; i < this.usedPlaces - 1; i++)
    {
        this.animalList[i] = this.animalList[i + 1];
    }
    this.animalList[this.usedPlaces - 1] = null;
    this.usedPlaces--;

    return releasedAnimal;
}

```

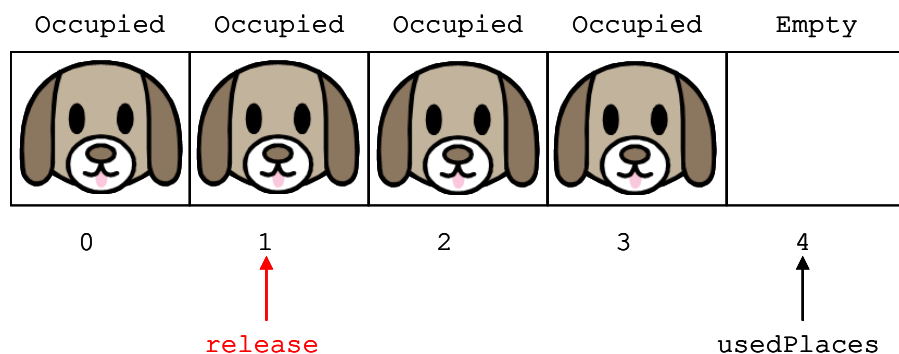
Shelter capacity (number of animals that can be accommodated) is set when the object is created. By default, it is the value of the constant **DefaultPlacesCount**. We use the field **usedPlaces** to monitor the occupied cells (at the same time we use it as index in the array for "pointing" to the first free place, counting from left to right in the array).



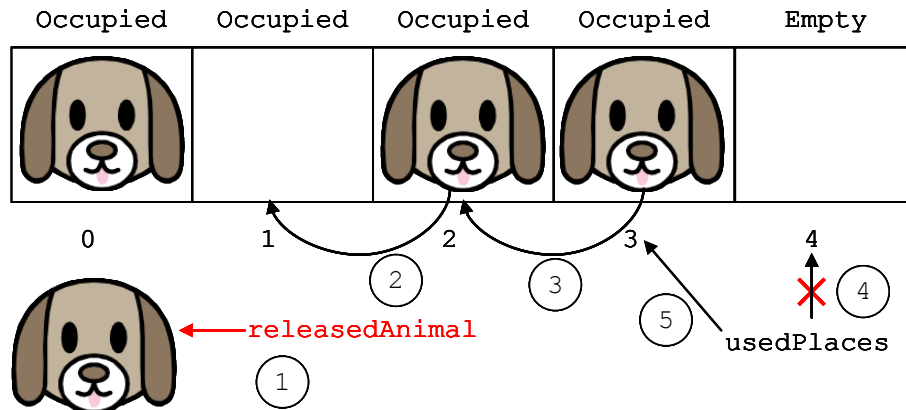
We have created a method for adding a new dog into the shelter – **Shelter()** and a method for releasing one from the shelter – **Release(int)**.

The method **Shelter()** throws an exception if there are no cells free, otherwise it adds a new animal to the first free cell, counting from the left side of the array.

The method **Release(int)** accepts the number of the cell from which a dog will be released (i.e. the index of the array, where a link is stored to the object of type **Dog** to be released).

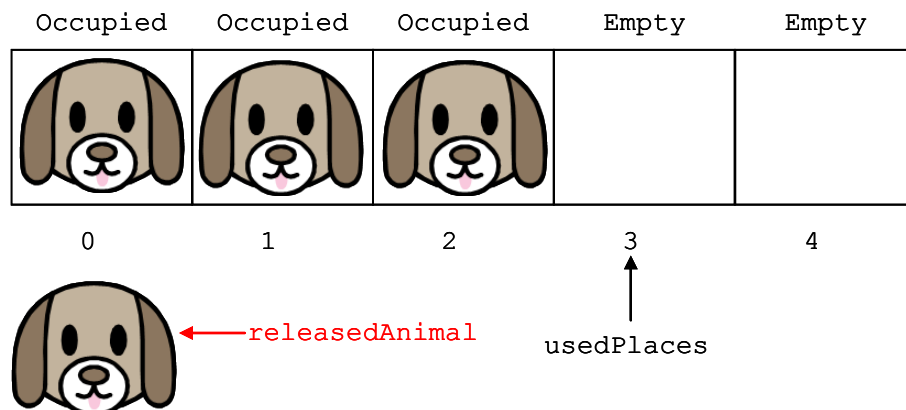


Then it moves all animals, which have a higher cell number then the emptied cell, one position to the left (steps 2 and 3 are shown in the diagram below).



The emptied cell at position **usedPlaces-1** is marked as free and a value of **null** is assigned to it. This erases the reference, allowing the system to clean the memory (garbage collector) for the object that is no longer used, preventing indirect loss of memory (memory leak).

Finally, the number of the last free cell is assigned to the **usedPlaces** field (steps 4 and 5 of the scheme above).



In case of many animals, the "removal" of an animal from a cell **could be a slow operation**, because it requires the transfer of all animals from adjacent cells one position to the left. In the chapter "[Linear Data Structures](#)", we will discuss more efficient ways of representing the animal shelter, but for now, we focus on the topic of generic types.

So far, we succeed in implementing the functionality of a shelter – the class **AnimalShelter**. When we work with objects of type **Dog** everything compiles and executes smoothly:

```
static void Main()
{
    AnimalShelter dogsShelter = new AnimalShelter(10);
    Dog dog1 = new Dog();
    Dog dog2 = new Dog();
    Dog dog3 = new Dog();

    dogsShelter.Shelter(dog1);
    dogsShelter.Shelter(dog2);
}
```

```
dogsShelter.Shelter(dog3);

dogsShelter.Release(1); // Releasing dog2
}
```

What happens, however, if we attempt to use an **AnimalShelter** class for objects of type **Cat**:

```
static void Main()
{
    AnimalShelter dogsShelter = new AnimalShelter(10);

    Cat cat1 = new Cat();

    dogsShelter.Shelter(cat1);
}
```

As might be expected, the **compiler displays an error**:

```
The best overloaded method match for 'AnimalShelter.Shelter(
Dog)' has some invalid arguments. Argument 1: cannot convert from
'Cat' to 'Dog'
```

Consequently, if we want to create a shelter for cats, we will not be able to reuse the class that we already created, although the operations of adding and removing animals from the shelter will be identical. Therefore, we have to literally copy the **AnimalShelter** class and change the type of the sheltered objects to – **Cat**.

Fine, but if we decide to provide shelter to other species, then how many classes of shelters for each type of animal do we have to create?

Our solution of the problem **is not sufficiently comprehensive** and does not fully meet the terms, which we set, namely– to have a **single class** describing a shelter suitable **for any kind of animal** (i.e. for all objects), while **it should contain only one kind of animal at a time** (i.e. only objects of one and the same type).

Instead of a particular type such as **Dog**, we could use a universal type **object**, which can take values as **Dog**, **Cat** and any other data type. However, this will create some inconvenience associated with the need to convert back, from **object** to **Dog**, when specifically creating a shelter for dogs that contains cells of type **object**, instead of type **Dog**.

To solve the task efficiently, we have to use a feature of the C# language that allows us to satisfy all required conditions simultaneously and that is called **generics** (template classes).

## What Is a Generic Class?

When a method needs additional information to operate properly, this information is passed to the method using parameters. During the execution of the program, when calling this particular method, we pass arguments to the method, which are assigned to its parameters and then used in the method's body.

When functionality (actions), encapsulated into a class, can be applied not only to objects of one, but many (heterogeneous) types, not known at the time of declaring the class; then we can use a functionality of the language C# that is called **generics** (generic types).

It allows us to **declare parameters of a class, by indicating an unknown type** that this class will work eventually with. When we instantiate a generic class, we replace the unknown type with a specified type. Accordingly, a newly created object will only work with objects of the type that we have assigned during initialization. The specific type can be any data type that the compiler recognizes, including class, structure, enumeration or even another generic class.

To get a clearer picture of the nature of the generic type, let's return to our example of the [previous section](#). The class that describes the animal shelter (**AnimalShelter**) **has to operate with different types of animals**. Therefore, if we want to create a general solution for the task of sheltering different animals, then we cannot define what type of animals are sheltered, during declaration of the class **AnimalShelter**. This means that we should make our class generic, adding to the declaration of the class a parameter of unknown type for different animals.

Later, when we want to create a dog's shelter for example, this parameter of the class will pass the name of our type – class **Dog**. Accordingly, if you create a shelter for cats, we will pass the type **Cat**, etc.



**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.**

In the following sections we will introduce the syntax of generic classes and we will modify our previous example to use generics.

## Declaration of Generic Class

Formally, the **parameterizing of a class** is done by adding **<T>** to the declaration of the class, after its name, where T is the substitute (parameter) of the type, to be used later:

```
[<modifiers>] class <class_name><T>
{
}
```

It should be noted that the angle brackets '<' and '>', which surround the substitution T, are an obligatory part of the syntax of C# and must occur in the declaration of a generic class.

The **declaration of a generic class**, which describes a shelter for homeless animals, should be as follows:

```
class AnimalShelter<T>
{
    // Class body here ...
}
```

We create a **template class AnimalShelter**, which we will further specify later, when replacing T with a specific type, for instance a type **Dog**.

A class may have more than one type parameter (to be substituted by more than one type), depending on need:

```
[<modifiers>] class <class_name><T1 [, T2, [... [, Tn]]]>
{
}
```

If the class needs **several different unknown types**, these types should be listed separated by a comma, between the angle brackets '<' and '>' in the declaration of the class. Each of the type parameters must have its unique identifier (e.g. a different letter) – in the definition indicated as **T1, T2, ..., Tn**.

In case, we should have to create a shelter for animals of mixed type, for instance one that accommodates both – dogs and cats, we might declare the class as follows:

```
class AnimalShelter<TDog, TCat>
{
    // Class body here ...
}
```

If this were our case, we might use the first parameter **TDog**, to indicate objects of type **Dog**, and a second parameter **TCat** – to indicate objects of type **Cat**.

## Specifying Generic Classes

Before we present more details about generics, we should look at **how to use generic classes**:

```
<class_name><concrete_type><variable_name> =
    new <class_name><concrete_type>();
```

Similar to the **T** type parameter in the declaration of a generic class, the angle brackets '<' and '>' have to surround the class **concrete\_type**.

For instance, if we create two shelters, one for dogs and one for cats, we should use the following code:

```
AnimalShelter<Dog> dogsShelter = new AnimalShelter<Dog>();
AnimalShelter<Cat> catsShelter = new AnimalShelter<Cat>();
```

In this way, we ensure that the shelter **dogsShelter** will always contain objects of type **Dog** and the variable **catsShelter** will always operate with objects of type **Cat**.

## Using Unknown Types by Declaring Fields

Once used during the class declaration, the type parameters that indicate the unknown types are visible to the whole body of the class. They can be used to declare a field of each type:

```
[<modifiers>] T <field_name>;
```

In our example of the shelter for homeless animals, we can use this feature of C# to declare the type of field **animalsList**, which holds references to objects for the housed animals, instead of a specific type of, for instance, **Dog**, using parameter **T**:

```
private T[] animalList;
```



When we create an object from our class, setting a specific type (e.g. **Dog**) during the execution of the program, **the unknown type T will be replaced** with that specific type. If we choose to create a shelter for dogs, we can consider that our field will be declared as follows:

```
private Dog[] animalList;
```

Accordingly, when we want to initialize a particular field in the constructor of our class, we should do it as usual – by creating an array, while using the parameter type – T:

```
public AnimalShelter(int placesNumber)
{
    animalList = new T[placesNumber]; // Initialization
    usedPlaces = 0;
}
```

## Using Unknown Types in a Method's Declaration

An **unknown type**, used in the declaration of a generic class, is visible from opening to closing curly bracket of the class body and **can be used in a method declaration**: As a parameter in the list of parameters of a method:

```
<return_type> MethodWithParamsOfT(T param)
```

- As a result of a method:

```
T MethodWithReturnTypeOfT(<params>)
```

Using our example, we adjust the methods **Shelter(...)** and **Release(...)**, respectively:

- To a method of unknown type parameter T:

```
public void Shelter(T newAnimal)
{
    // Method's body goes here ...
}
```

- And to a method, which returns a result of unknown type T:

```
public T Release(int i)
{
    // Method's body goes here ...
}
```

When creating an object from our class shelter and replacing the unknown type with a specific one (e.g. **Cat**) during execution of the program, above methods will have the following form:

- The parameter of method **Shelter** will be of type **Cat**:

```
public void Shelter(Cat newAnimal)
{
```

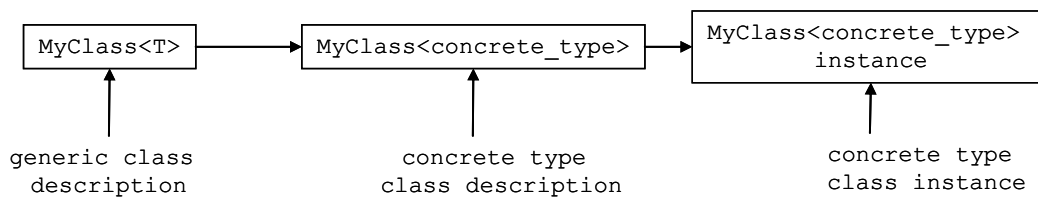
```
}  
    // Method's body goes here ...  
}
```

- The method **Release** will return a result of type **Cat**:

```
public Cat Release(int i)  
{  
    // Method's body goes here ...  
}
```

## Generics – Behind the Scenes

Before we continue, let's explain what happens in the memory of the computer, when we work with generic classes.



First we declare a generic class **MyClass<T>** (generic class description in the scheme above). Then, the compiler translates our code into intermediate language (MSIL). The translated code contains information about the class being generic, i.e. it still works with undefined types. At runtime, when the generic class is used with a specific type, then a new **description of the class** is created (specific type class description in the diagram above), which is identical to the generic class, except that **T** is replaced by a specific type. For example, if you use **MyClass<int>**, then everywhere in your code the unknown parameter **T** will be replaced with **int**. After that, we can create an object of a generic class with type **int**. Note that, to create this object, the intermediate, concrete type, class description will be used. Instantiating of a generic class with specific types of its parameters is called "**specialization of the type**" or "**extension of a generic class**".

Using our example, if we create an object of type **AnimalShelter<T>**, which works only with objects of type **Dog**, adding an object of type **Cat** causes a compile error almost identical to the errors that occurred during the attempt to add an object of type **Cat** into an object of type **AnimalShelter** of the section "[Shelter for Homeless Animals – Example](#)":

```
static void Main()  
{  
    AnimalShelter<Dog> dogsShelter = new  
    AnimalShelter<Dog>(10);  
  
    Cat cat1 = new Cat();  
  
    dogsShelter.Shelter(cat1);  
}
```

As expected, we get the following **compilation error messages**:

The best overloaded method match for 'AnimalShelter<Dog>.Shelter(Dog)' has some invalid arguments

Argument 1: cannot convert from 'Cat' to 'Dog'

## Generic Methods

Like classes, when the type of the method's parameters cannot be specified, we can **type parameterize the method**. Subsequently, the implementation of a specific type will happen during the invocation of the method, replacing the unknown type with the specific one, as for classes.

**Type parametrization of a method** is done by adding **<K>** after the name and before the opening bracket of the method. Where **K** is any arbitrary character that is not **T**, substituting for the type that will be used later:

```
<return_type><methods_name><K>(<params>)
```

Accordingly, we can use undefined type **K** for parameters in the parameter list of the method **<params>** and also for the return value or, to declare variables of substitute type **K** in the body of the method.

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 caring for their types**, so that we can use it for all types of variables.

If we want to swap the values of two integers and two string variables, then we use the method as follows:

```
int num1 = 3;
int num2 = 5;
Console.WriteLine("Before swap: {0} {1}", num1, num2);
// Invoking the method with concrete type (int)
Swap<int>(ref num1, ref num2);
Console.WriteLine("After swap: {0} {1}\n", num1, num2);

string str1 = "Hello";
string str2 = "There";
Console.WriteLine("Before swap: {0} {1}!", str1, str2);
// Invoking the method with concrete type (string)
Swap<string>(ref str1, ref str2);
Console.WriteLine("After swap: {0} {1}!", str1, str2);
```

When you run this code, the result is:

```
Before swap: 3 5
After swap: 5 3
```

```
Before swap: Hello There!
After swap: There Hello!
```

Note that, in the list of parameters we have used also the keyword **ref**. This concerns the specification of the method – namely, to pass the values of two references. By using the keyword **ref**, the method will use the same reference that was given by the calling method. In this way, a variable, changed by our method is preserved after exiting the method.

When **calling a generic method**, we can omit the declaration of the type (in our example `<int>`), because the compiler will infer it automatically, recognizing the type of the given parameters. In other words, our code can be simplified using the following calls:

```
Swap(ref num1, ref num2); // Invoking the method Swap<int>
Swap(ref str1, ref str2); // Invoking the method Swap<string>
```

The **compiler will recognize what is the specific type**, however, only if this type is part of the parameter's list. The compiler cannot recognize the specific type of a generic method only by the type of its return value, when the method has no parameters. In that case, the specific type must be provided explicitly. In our example, it will be similar to the original method call, or by adding `<int>` or `<string>`.

Static methods can also be made generic, unlike properties and constructors of the class.



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

## Features of Declaration of Generic Methods in Generic Classes

As we have seen in the section "[Using Unknown Types in a Declaration of Methods](#)", non-generic methods can use unknown types, when defined as parameter in the generic class declaration (e.g. methods **Shelter()** and **Release()** from the example "Shelter for Homeless Animals"):

### AnimalShelter.cs

```
public class AnimalShelter<T>
{
    // ... The rest of the code ...

    public void Shelter(T newAnimal)
    {
        // Method body here
    }

    public T Release(int i)
    {
        // Method body here
    }
}
```

```
}
```

If we try to reuse the variable, which represents the unknown type of the generic class, for example as **T** in the declaration of a generic method, then we will get the warning **CS0693**. This happens because the scope of the unknown type **T**, defined in the declaration of the method, overlaps the scope of the unknown type **T**, defined in the declaration of the class:

CommonOperations.cs
<pre>public class CommonOperations&lt;T&gt; {     // CS0693     public void Swap&lt;T&gt;(ref T a, ref T b)     {         T oldA = a;         a = b;         b = oldA;     } }</pre>

When you try to compile this class, you receive the following **message**:

Type parameter 'T' has the same name as the type parameter from outer type 'CommonOperations<T>'
--------------------------------------------------------------------------------------------------

So, if we want our code to be flexible and our generic method safely called with a specific type, different from that of the generic class, we have to name the parameter of the unknown type in the declaration of the generic method **differently than the parameter of the unknown type** in the class declaration, as shown below:

CommonOperations.cs
<pre>public class CommonOperations&lt;T&gt; {     // No warning     public void Swap&lt;K&gt;(ref K a, ref K b)     {         K oldA = a;         a = b;         b = oldA;     } }</pre>

Thus, always make sure that there will be no overlapping of parameters of the unknown types with the same name for method and class.

## Using the Keyword "default" in a Generic Source Code

Now that we have introduced the basics of generic types, let's try to **redesign our first example** from the earlier section ([Shelter for Homeless Animals](#)). The only thing we need to do is to replace the type **Dog** with type parameter **T**:

## AnimalsShelter.cs

```
public class AnimalShelter<T>
{
    private const int DefaultPlacesCount = 20;

    private T[] animalList;
    private int usedPlaces;

    public AnimalShelter() : this(DefaultPlacesCount)
    {
    }

    public AnimalShelter(int placesCount)
    {
        this.animalList = new T[placesCount];
        this.usedPlaces = 0;
    }

    public void Shelter(T newAnimal)
    {
        if (this.usedPlaces >= this.animalList.Length)
        {
            throw new InvalidOperationException("Shelter
is full.");
        }
        this.animalList[this.usedPlaces] = newAnimal;
        this.usedPlaces++;
    }

    public T Release(int index)
    {
        if (index < 0 || index >= this.usedPlaces)
        {
            throw new ArgumentOutOfRangeException(
                "Invalid cell index: " + index);
        }
        T releasedAnimal = this.animalList[index];
        for (int i = index; i < this.usedPlaces - 1; i++)
        {
            this.animalList[i] = this.animalList[i + 1];
        }
        this.animalList[this.usedPlaces - 1] = null;
        this.usedPlaces--;

        return releasedAnimal;
    }
}
```

Everything should work properly, however, if we try to compile the class, then we **get the following error**:

Cannot convert null to type parameter 'T' because it could be a non-nullable value type. Consider using 'default(T)' instead.

The error appears to be inside the method **Release()** and it is related to the recording of a **null** value in the last released (rightmost) cell of the shelter. The problem is that we are trying to use the default value for a reference type, but **we are not sure whether this type is a reference type or a primitive**. Therefore, the compiler displays the above error. If the type **AnimalShelter** is instantiated by a structure and not by a class, then the null value is not valid.

To handle this problem, 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 a reference type is **null** and for numeric types it is zero. We make the following change:

```
// this.animalList[this.usedPlaces - 1] = null;
this.animalList[this.usedPlaces - 1] = default(T);
```

Now the compilation runs smoothly and the class **AnimalShelter<T>** operates correctly. We can test this as follows:

```
static void Main()
{
    AnimalShelter<Dog> shelter = new AnimalShelter<Dog>();
    shelter.Shelter(new Dog());
    shelter.Shelter(new Dog());
    shelter.Shelter(new Dog());
    Dog d = shelter.Release(1); // Release the second dog
    Console.WriteLine(d);
    d = shelter.Release(0); // Release the first dog
    Console.WriteLine(d);
    d = shelter.Release(0); // Release the third dog
    Console.WriteLine(d);
    d = shelter.Release(0); // Exception: invalid cell index
}
```

## Advantages and Disadvantages of Generics

Generic classes and methods **increase the reusability of the code**, its security and its performance, as 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 the level of abstraction in the program and the more flexible and reusable source code becomes. However, 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.

## Naming the Parameters of the Generic Types

Before we complete the topic of generics, let's look at some guidance on working with parameters of unknown types in a generic class:

1. If there is just one unknown type in the generic class, it is common to use the letter **T**, as a parameter for that unknown type. The class declaration **AnimalShelter<T>**, which we used before, serves as an example.
2. The parameters for unknown types should have most descriptive names, as this will improve the readability of the source code. For instance, we can modify our example, replacing the letter **T**, with a more descriptive parameter of unknown type that, for instance, is called **Animal**:

AnimalShelter.cs
<pre>public class AnimalShelter&lt;Animal&gt; {     // ... The rest of the code ...      public void Shelter(Animal newAnimal)     {         // Method body here     }      public Animal Release(int i)     {         // Method body here     } }</pre>

When we use descriptive names for parameters of unknown type instead of a letter, then it is better to add **T** at the beginning of the name, to better distinguish it from the class names in our application. In other words, instead of using **Animal** in the previous example, we should use **TAnimal** (**T** comes from the word "template" which means a parameterized / generic type).

## Exercises

1. Define a class **Student**, which contains the following **information about students**: full name, course, subject, university, e-mail and phone number.
2. Declare several **constructors** for the class **Student**, which have different lists of parameters (for complete information about a student or part of it). Data, which has no initial value have to be initialized with **null**. Use nullable types for all non-mandatory data.
3. Add a **static field** to the class **Student**, which holds the number of created objects of this class.
4. Add a **method** to the class **Student**, which displays complete information about the student.
5. Modify the source code of the **Student** class, so as to **encapsulate** the data in the class, using **properties**.



6. Write a class **StudentTest**, which has to **test the functionality** of the class **Student**.
7. Add a **static method** to the class **StudentTest**, which creates several objects of type **Student** and that stores them in static fields. Create **static properties** for the class to access these fields. Write a test program, which displays the information about these fields in the console.
8. Define a class which contains information about a **mobile phone**: model, manufacturer, price, owner, features of the battery (model, idle time and hours talk) and features of the screen (size and colors).
9. Declare several **constructors** for each of the classes created in the previous tasks, which have different lists of parameters (for complete information about a student or part of it). Data fields that are unknown have to be initialized respectively with **null** or **0**.
10. To the class of mobile phone in the previous two tasks, add a **static field nokiaN95**, which stores information about mobile phone model Nokia N95. Add a method to the same class, which displays information about this static field.
11. Add an **enumeration BatteryType**, which contains the values for type of the battery (Li-Ion, NiMH, NiCd, ...) and use it as a new field for the class **Battery**.
12. Add a method to the class **GSM**, which returns information about the object as a **string**.
13. Define properties to encapsulate the data in classes **GSM**, **Battery** and **Display**.
14. Write a class **GSMTest**, which has to **test the functionality** of class **GSM**. Create few objects of the class and store them into an array. Display information about the created objects. Display information about the static field **nokiaN95**.
15. Create a class **Call**, which contains information about a call made via mobile phone. It should contain information about date, time of start and duration of the call.
16. Add a property for keeping a **call history** – **CallHistory**, which holds a list of call records.
17. In **GSM** class add methods for adding and deleting calls (**Call**) in the archive of mobile phone calls. Add a method which deletes all calls from the archive.
18. In the **GSM** class, add a method that calculates the total amount of calls (**Call**) from the archive of phone calls (**CallHistory**), as the price of a phone call is passed as a parameter to the method.
19. Create a class **GSMCallHistoryTest**, with which to test the functionality of the class **GSM** from task 12, as an object of type **GSM**. Then add to it a few phone calls (**Call**). Display information about each phone call. Assuming that the price per minute is 0.37, calculate and display the total cost of all calls. Remove the longest conversation from archive with phone calls and calculate the total price for all calls again. Finally, clear the archive.
20. There is a **book library**. Define classes respectively for a **book** and a **library**. The library must contain a name and a list of books. The books must contain the title, author, publisher, release date and ISBN-number. In the class, which describes the library, create methods to add a book to the library, to search for a book by a predefined author, to display information about a book and to delete a book from the library.
21. Write a **test class**, which creates an object of type library, adds several books to it and displays information about each of them. Implement a test functionality, which finds all books authored by Stephen King and deletes them. Finally, display information for each of the remaining books.

22. We have a **school**. In the school we have **classes** and **students**. Each class has a number of **teachers**. Each teacher teaches a variety of disciplines. Students have a name and a unique number in the class. Classes have a unique text identifier. Disciplines have a name, a number of lessons and a number of exercises. The task is to model the school with C# classes. You have to define the classes with their fields, properties, methods and constructors. Also **define a test class**, which demonstrates that the other classes work correctly.
23. Write a **generic class GenericList<T>**, which holds a list of elements of type **T**. Store the list of elements into an array with a limited capacity that is passed as a parameter to the constructor of the class. Add methods to add an item, to access an item by index, to remove an item by index, to insert an item at given position, to clear the list, to search for an item by value and to override the method **ToString()**.
24. Implement **auto-resizing functionality** for the array from the previous task, when by adding of an element, the capacity of the array is reached.
25. Define a class **Fraction**, which contains information about the **rational fraction** (e.g.  $\frac{1}{4}$  or  $\frac{1}{2}$ ). Define a static method **Parse()** to create a fraction from a sting (for example **-3/4**). Define the appropriate properties and constructors for the class. Also add a property of type **Decimal** to return the decimal value of the fraction (e.g. 0.25).
26. Write a class **FractionTest**, which tests the functionality of the class **Fraction** from the previous task. Pay close attention to the testing of the function Parse usinz different input data.
27. Write a function to **cancel a fraction** (e.g. if numerator and denominator are respectively 10 and 15, the fraction will be cancelled to 2/3).

## Solutions and Guidelines

1. Use **enum** for subjects and universities.
2. To avoid repetition of source code, call **constructors** from each other with keyword **this(<parameters>)**.
3. Use the constructor of the class for increasing the number of objects created from class **Student**.
4. Display on the console each field of the class **Student**, separated by a blank line.
5. Define as **private** all members of the class **Student** and then use Visual Studio (Refactor -> Encapsulate Field) to automatically define the public **get** / **set** methods for accessing these fields.
6. **Create a few students** and display the entire information for each one of them.
7. You can use the **static constructor** to create instances during first access to the class.
8. Declare three separate classes: **GSM**, **Battery** and **Display**.
9. Define the described constructors and **create a test program** to check if classes are working properly.
10. Define a **private** field and initialize it at the time of its declaration.
11. Use **enum** for the **type of battery**. Search in Internet for other types of batteries for phones, accept these in the requirements and add them as value of the enumeration.
12. Override the method **ToString()**.

13. In classes **GSM**, **Battery** and **Display**, define suitable **private** fields and generate **get / set**. You can use automatic generation in Visual Studio.
14. Add a method **PrintInfo()** to class **GSM**.
15. Read about the class **List<T>** in Internet. The class **GSM** has to store its conversations in a list of type **List<Call>**.
16. Return as a result the **list of conversations**.
17. Use the built-in methods of the class **List<T>**.
18. Because the **tariff is fixed**, you can easily **calculate the total price** of all calls.
19. **Follow the instructions** directly from the requirements of the task.
20. Define classes **Book** and **Library**. For a list of books use **List<Book>**.
21. Follow the instructions directly from the requirements of the task.
22. Create classes **School**, **SchoolClass**, **Student**, **Teacher**, **Discipline** and define for them their respective fields, as described in the instructions of the task. Do not use the word "**Class**" as a class name, because in C# it has special meaning. Add methods for printing all the fields from each of the classes.
23. Use your knowledge concerning **generic classes**. Check out all input parameters of the methods, just to make sure that no element can access an invalid position.
24. When you reach the capacity of the array, **create a new array with a double size** and copy all old elements in the new one.
25. Write a class with two **private decimal** fields, which hold information relevant to the **numerator** and **denominator** of the fraction. Among other requirements in the task, redefine in appropriate standard, the features for each object: **Equals(...)**, **GetHashCode()**, **ToString()**.
26. Figure out appropriate **tests**, for which your function may give incorrect results. Good practice is **first to write the tests**, then to implement their specific functionality.
27. Search for information in Internet for the "**greatest common divisor (GCD)**" and the **Euclidean algorithm** for its calculation. Divide the numerator and denominator by their greatest common divisor and you will get the cancelled fraction.