Chapter 14. Defining Classes

In This Chapter

In this chapter we will understand how to **define custom classes** and their elements. We will learn 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 observe the characteristics of the constructors and we will set 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 – 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

The aim of every program written by the programmer is to **solve a given problem** based on the implementation of a certain idea. In order to create a solution, first, we sketch a simplified actual model, which does not represent everything, but focuses on these facts, which are significant for the end result. Afterwards, based on the sketched model, we are looking for an answer (i.e. to create an algorithm) for our problem and the solution we describe via given programming language.

Nowadays, the most used programming languages are the object-oriented. And because the **object-oriented programming (OOP)** is close to the way humans think, using one easily allows us to describe models of the surrounding life. Certain reason for this behavior is, because OOP offers tools to draw the set of concepts, which outline classes of objects in every model. The term – class and the definition of custom classes, different from the .NET system framework's, is built-in feature of the C# programming language. The purpose of this chapter is to get us know with it.

Let's Recall: What Does It Mean Class and Object?

Class in the OOP is called a definition (**specification**) of a given type of objects from the real-world. The class represents a pattern, which describes the different states and behavior of the certain objects (the copies), which are created from this class (pattern).

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

For example, if we have a class type **Dog**, which describes some of the characteristics of a real dog, then, the objects based on the description of the class (e.g. the doggies "Fido" and "Rex") are from type class **Dog**. It means the same when the string "some string" is from class type **String**. The difference is that **objects** from type **Dog** is are copies of the 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 what kind of data types and objects has in order to be described. The object (the certain copy of this class) holds the actual **data**. The data defines the object's **state**.

In addition to the **state**, in the class is described the **behavior** of the objects. The behavior is represented by actions, which can be performed by the objects themselves. The resource in OOP, through which we can describe this behavior of the objects from a given class, is the declaration of **methods** in the class body.

Elements of the Class

Now, we will go through the main elements of every class, and we will explain them in details latter. The main **elements of a C# classes** are the following:

- **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 idioms in the language, the classes also have single class body. It is defined right after the class declaration, enclosed in curly brackets "**{**" and "**}**". The content inside the brackets is known as body of the class. The elements of the class, which are numbered below, are part of the body.

```
public class Dog
{
  // … 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 (somewhere in the literature are known as **member-variables**). The data of the object, which these variables represent, and are retained into them, is the specific state of an object, and one is required for the proper work of object's methods. The values, which are in the fields, reflect the specific state of the given object, but despite of this there are other types of fields, called **static**, which are shared among all the objects.

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

- **Properties** – this is the way to describe the **characteristics** of a given class. Usually, the value of the characteristics is kept in the fields of the object. Similar to the fields, the properties may be held by certain object or to be shared among the rest of the 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 behavior based on the class type. Methods execute the implemented programming logic (algorithms) and the handling of data.

Sample Class: Dog

Here is how a class looks like. The class **Dog** defined here owns all the elements, which we described so far:

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

   // Field declaration
   private string name;

   // Constructor declaration (peremeterless 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
```

At the moment we will not explain in greater details this code, because the related information will be presented later in this chapter.

Usage of Class and Objects

In the chapter "Creating and Using Objects" we saw in details how new objects of a given class are created and how they can be used. Now, shortly we will revise this programming technique.

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

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

If we want to manipulate the newly created object, we will have to assign it to a variable from its class type. By doing it, in this variable we will keep the connection (reference) to the object.

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

Example – A Dog Meeting

Let’s have the example from the previous section where we defined the class **Dog**, describing a dog, and let’s add a method **Main()** to the class. In this method we will demonstrate how to use the mentioned elements until here: create few **Dog** objects, assign properties to these objects 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 wit 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 the execution 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 the example program, with the help of **Console.ReadLine()**, we got the name of the objects of type dog, which the user should input.

We assigned the first entered string to the variable **firstDogName**. Afterwards we used this variable when we created the first object from class type **Dog** – **firstDog**, by assigning it to the parameter of the constructor.

We created the second object **Dog**, without using a string for the name of the dog in the constructor. With the help of **Console.ReadLine()** we got the

name of the dog and then the value was assigned to the property **Name**. This is done by using a “dot” convention, applied to the variable, which keeps the reference to the second object from type **Dog** – **secondDog.Name**.

When we created the third object from class type **Dog**, we used for the name of the dog its default value which is **null**. Note that in the **Bark()** method dogs whthout name (**name == null**) are printed as “[unnamed dog]”.

Afterward we created an array from type **Dog**, by initializing it with the three newly created objects.

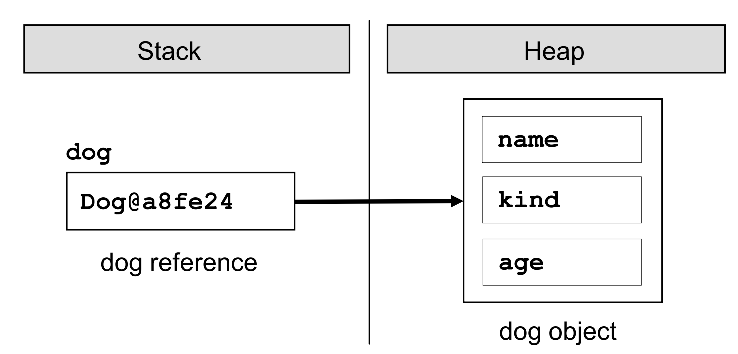
At the end, we used a loop, to go through the array of objects from type **Dog**. For every element from the array we again used the “dot” notation, be calling the method **Bark()** for the particular object: **dog.Bark()**.

Nature of Objects

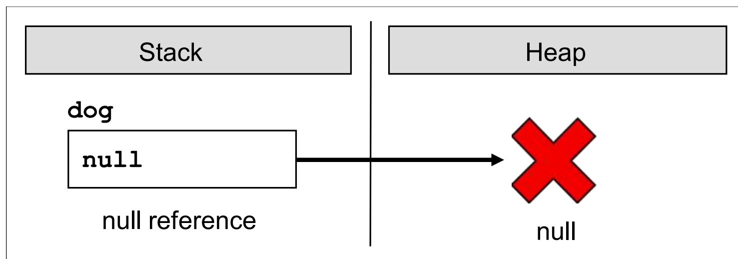
Let’s revise, when we create an object in .NET, one consists from two parts – the **significant part (data)**, which contains its data and it is located in the memory of the operating system called a dynamic memory (heap) and a **reference part** to this object, which resides in the other part of the operating system’s memory, where are stored the local variable and parameters of the methods (the program execution stack).

For example, let’s have a class called **Dog**, which has the properties for name, kind and age. Let’s create a variable **dog** from this class. This variable is a reference to the object and is in the dynamic memory (heap).

The **reference** is a variable, which can access objects. The figure below depicts an example reference, which has link to the real object in the heap, and is called with the name **dog**. One, compare to the variable from primitive (value type), does not contain the real value (i.e. the data of the object), but the address, where one is located in the heap memory:



When we declare one variable from type a particular 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 (there is a missing value):



Organizing Classes in Files and Namespaces

In C# the only one limitation regarding the saving of our own custom classes is: they 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 **every class to be stored in exactly one file, which corresponds to its name**, i.e. the class **Dog** should be saved in a file **Dog.cs**.

Organizing Classes in Namespaces

As we should know from the chapter "Creating and Using Objects", the **namespaces in C# are named group of classes**, which are logically connected, without a requirement how they are stored in the file system.

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

After the insertion of the used namespaces, the next is the declaration of the **namespace** of the classes in the file. As we know, there is no requirement to declare classes in a namespace, but it is a good programming technique if we do it, because the class distribution in the namespace is used for better organization of the code and determination of the classes with equal names.

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

The **full name of a class** in .NET Framework is the class name, preceded by the namespace in which the class is declared, e.g.: **<namespace_name>.<class_name>**. By the **using** reserved word we can use types from certain namespace, without writing the full name, e.g.:

```
using System;
…
DateTime date;
```

Instead of:

```
System.DateTime date;
```

One typical declaration sequence, which we should follow when we create 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 is already explained in the chapter "Creating and Using Objects" and therefore we will not discuss it again.

Before we continue, let’s look into the first line of the previous snippet. Instead include of namespace it is a source code comment. This is not a problem in compilation time, the comments are "removed" from the code and thus the first line is still the including statement.

Encoding of Files and Using of Cyrillic and Unicode

While we are 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 it is possible to use characters in our code from alphabets other than Latin. In the next example we use Cyrillic letters for identifiers in Bulgarian language as well as comments in the code, written in Bulgarian (in Cyrillic letters):

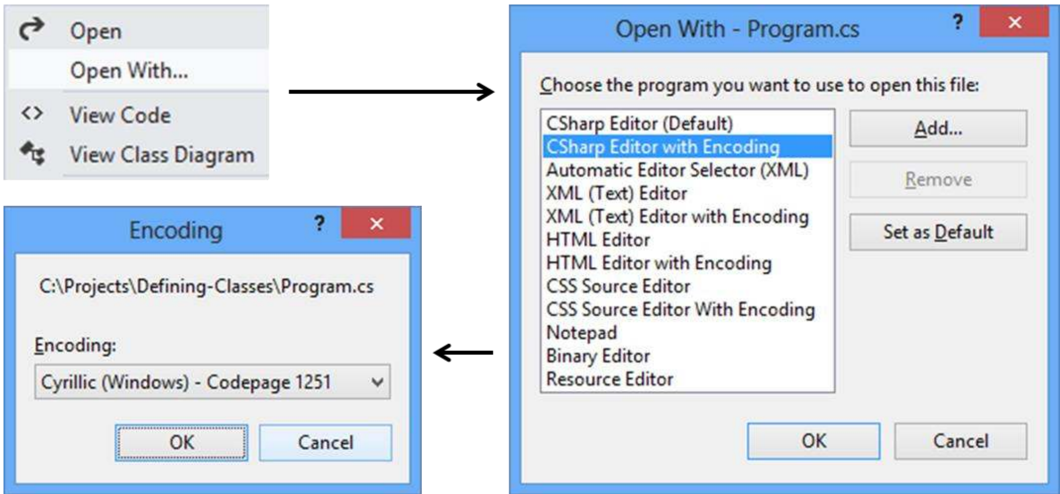
```
using System;

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, but to keep the characters readable in the Visual Studio editor we need to provide an appropriate **encoding of the file**.

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

To use a different encoding other than the system's default encoding in Visual Studio, we need to choose the appropriate encoding of the file when 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 in the file system with a specific encoding 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 this way our code will be readable for more people in the world.

Imagine that you live in Germany and you need to type 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 will handle variables and comments in German.

Modifiers and Access Levels (Visibility)

Let's revise, 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 and 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 the following elements of the class: class declaration, fields, properties and methods.

Modifiers and Access Levels

As we explained, in C# there are four access modifiers – **public**, **private**, **protected** and **internal**. Based on them we control the access (visibility) to 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 details only **public**, **private** and **internal**. More about **protected** and **protected internal** we will learn in "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**, no matter from the current project (assembly), from the current namespace. The access level **public** defines the miss of restrictions regarding the visibility. This access level is the least restricted access level in C#.

Access Level "private"

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

Access Level "internal"

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

Assembly

.NET assemblies are **collections of compiled types** (classes and other types) and **resources**, which form a logical unit. Assemblies are stored in a binary file of type **.exe** or **.dll**. All types in C# and as general in .NET Framework can reside only inside assemblies. By every compilation of a .NET application one or several assemblies are created by the C# compiler and each assembly is stored inside an **.exe** or **.dll** file.

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 stay the name of the class **<class_name>**.

Besides the reserved word **class** and the name of the class, in the declaration of the class can be used several modifiers, e.g. the reviewed until now modifiers.

Class Visibility

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

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

There is also another operation, which can be done over the classes, when the visibility allows it. The operation is called **inheritance of a class**, but we will discuss it later in the chapter Object-Oriented Programming Principles.

As we understood, the access level term means "**visibility**". If the class **A** cannot "see" the class **B**, the access level of the methods and the fields in **B** does not matter.

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

Access Level "public"

If we declare a class access modifier as **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 the public class.

Just to know, if we want to use a class with access level **public** from other namespace, different from the current, we should use the reserved word for including different namespaces **using** or every time we should write the full name of the class.

Access Level "internal"

If we declare one class with access modifier **internal**, one will be **accessible only from the same namespace**. It means that only the classes from the same assembly can create objects from this type class and to have access to the methods and fields (with related access level) of the class. This access level is the default, where it is not used access modifier by the declaration of the class.

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

Access Level "private"

If we want to be exhaustive, we have to mention that as access modifier for a class can be used the visibility modifier **private**, but this is related to the term "inner class" (nested class), which we will review in the "Nested Classes" section. Private classes like other private members are accessible only inside the class which defined 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 opening curly bracket "**{**" and ends with closing one – "**}**". The class always should have a body.

Class Naming Convention

Equal to the methods, for creation of the class names there are the following common standards:

1. The names of the classes begin with capital letter, and the rest of the letters are lower case. If the name of the class consists of several words, every word begins with capital letter, without separator to be used. This is the well-known **PascalCase** convention.
2. For name of the classes **nouns** are usually used.
3. It is recommended the name of the class to be in **English** language.

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

```
Dog
Account
Car
BufferedReader
```

More about the name of the classes we will learn in the chapter "High-Quality Programming Code".

The Reserved Word "this"

The reserved word **this** in C# is used to **reference the current object**, when one is used from method in the same class. This is the object, which method or constructor is called. The reserved word can be deemed as an

address (reference), given priory from the language authors, with which we access the elements (fields, methods, constructor) of the own class:

```
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
```

Currently, we will not explain the given code above. Later, we will do it in other sections of this chapter, dedicated to the elements of the class (fields, methods, constructors) and as well related to 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 in 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 different 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 block).
2. **Parameters** – these are the variables in the list of parameters, which one method can have.

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

Fields are declared in the body of the class, outside the body of a single method or constructor.

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, the declaration of a field is done in the following way:

```
[<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 the name of the normal variables, when we declare the name of the instance-variables, we should obey the rules for naming of identifiers in C# (see chapter "Primitive Types and Variables").

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

Modifiers and the access modifiers, allowed in the declaration of one field, are explained in chapter "Primitive Types and Variables".

In this chapter, from the other modifiers, which are not based on access levels, and can be used in the declaration of fields, we will discuss **static**, **const** and **readonly**.

Scope

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

Initialization during Declaration

When we declare one field it is possible to assign to it an initial value. We do this similarly to an assignment of normal local variable:

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

Of course, the **<initial_value>** has to be a type compatible with the field's type, e.g.:

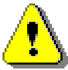
```
class SampleClass
{
    int age = 5;
    long distance = 234; // The literal 234 is of integer type

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

```

```
    // … Other code …
}
```

Default Values of the Fields

Every time, when we create a new object of a given class, it is allocated memory in the heap for every field from the class. In order this to be done the memory is **initialized automatically with the default values** for the certain field. The fields, which do not have explicitly a default value in the code, use the default value specified for the .NET type, to which they belong.

This is different for the 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, except if they are not explicitly initialized with some other value.

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

The value of all types is 0 or something similar. 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 for it fields **name**, **age** and **length** and check for the gender **isMale**, without explicitly initializing them, they will be automatically zeroed when we create an object of this class:

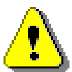
```
public class Dog
{
   string name;
   int age;
   int length;
   bool isMale;

   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);
   }
}
```

Respectively, when we execute the program we will have as output the following:

```
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 one method, without initializing it, and afterward we try to use it (e.g. printing its value), this will trigger a **compilation error**, because the local variables are not initialized with default values when they are declared.



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

Let's have look into one 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

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

One example for such initialization is the modified example class **SampleClass** from 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 in this section, in the declaration of one field is allowed to use the modifications **const** and **readonly**. The fields, declared as **const** or **readonly** are called **constants**. They are used when a certain **value is used several times**. These values are declared only ones without repetitions. Examples of constants in the .NET Framework are the mathematical constants **Math.PI** and **Math.E**, and as well the constants **String.Empty** and **Int32.MaxValue**.

Constants Based on "const"

The fields, declared with **const**, have to be initialized during the de facto declaration and afterwards theirs value cannot be changed. They can be accessed without to create an instance (an object) of the class and they are common for all created objects in our program. Something more, when we compile the code, the places where **const** fields are referred are replaced with theirs particular values directly without to use the constant variable at all. For this reason the **const** fields are called **compile-time constants**, because they are replaced with the value during the compilation process.

Constants Based on "readonly"

The modifier **readonly** creates fields, which values cannot be changed once they are assigned. Fields, declared as **readonly**, allow one-time initialization either in the moment of the declaration or in the class constructors. Later theirs values cannot be changed. Because of this reason, the **readonly** fields are called **run-time constants** – constants, because their values cannot be changed after assignment and run-time, because this process happens during the execution of the program (in runtime).

Let's illustrate the foregoing 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 chapter "Methods" we have discussed how to **declare and use a method**. In this section we will revise how we do this and we will focus on some additional features from the process of creating methods. Till now we have used static methods only. Now it is time to start using non-static (instance) methods.

Declaring of Class Method

The declaration of methods is done in the following way:

```
// 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, which we declare, when this is required.

We know, if the return type **<return_type>** is **void**, then **<return_statement>** can be declared without the **return** statement. If **<return_type>** is different from **void**, the method has to return a result with the help of the reserved word **return** and an expression, which is from the type **<return_type>** or a compatible one.

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

We already discussed some of the access modifiers that can be used in the declaration of a method in the section "Visibility of Methods and Fields" we will review in details this again.

The **static** modifier will be explained in depth in the section "Static Classes and Static Members".

Example – Method Declaration

Let's see 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 based on the "dot" operator we can access fields and to call the methods of a given class. Now, let's recall how we use conventional non-static methods of a given class, i.e. the methods do not have the modifier **static** in theirs declaration.

E.g. let's have the class **Dog** with the field **age**. To print the value of this field we need to create a **Dog** instance and 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 access to the value of one field can be done 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, from 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 from the **Main()** method to access the values of the field **age** of the object **dog**, we simple call the method **GetAge()**:

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

The result of the execution based on the change will be the same.

Formally, the declaration of access to a field in 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 block, which is without **static** modifier.

Except for retrieving of the value of one 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 this method will increment the age with one year:

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

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

```
// 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 **static** modifier in theirs declarations, the methods, which are also non-static, can be called in the body of a class via the reserved word **this**. This is happening again with the "dot" notation and more specifically with the required arguments (if there are any):

```
this.<method_name>(…)
```

For example, let's create a method **PrintAge()**, which prints the age of the object from type **Dog**, and for this purpose calls the method **GetAge()**:

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

The first line of the example is indicating that we want to receive the age (the value of the field **age**) of 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".

Skip "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 like this.age
}

public void MakeOlder()
{
   age++; // The same like 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 skipped and directly to access the elements of the class.

Although it is understood clearly, the reserved word **this** is often used for access to fields in the class, because it helps to make 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 skipped 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 one field** starts from the line where the declaration is made to the closing curly bracket of the class. For example let's see 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
```

On the other hand, when we implement the body of one method we have to declare local variables which we will use for the work of the method. As we know, the **scope of a local variable** begins from the line where it is declared to the closing bracket of the body of the method. For example, let's add this 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 overlap with the name of some field. In this case there is a collision.

Let's first look at one example, before we explain what it is 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 in this way the method, could it be possible to compile this code? And if it is compiled, is it possible to execute it? If it is compiled and executed 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 names match with fields of the class**. If this happens, we say that the scope of the local variable overlays the field variable (**scope overlapping**).

Therefore the scope of the local variable **myValue** with value **5** overlapped the scope of the field variable in the class. Then, when we print we will get the local variable value.

Despite this, sometimes it is required use the field instead the local variable with the same name. In this case, to retrieve the value of the field, we use the reserved word **this**. For this purpose we access the field by using the "dot" operator, applied to the reserved word **this**. In this way, we say deliberately 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 relate to the printing of the value **myValue**:

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

This time, after we applied 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 generality of the **modifiers and the access levels** for the elements in one class in C#. Later we have discussed the access level in the declaration for one 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 expose these rules simultaneously.

Differently 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 have access) from other class **B**, then none of its elements (fields and method) can be accessed from class **B**.

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

In the next subsections, to the explanations until now, we will review examples, in which we have two classes (**Dog** and **Kid**) and which are visible one to other, i.e. every from the classes can create objects from the other type – the other class and to access its elements depending from the defined access level declared. 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 the 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();
    }
}
```

Currently, all elements (fields and methods) of both classes are declared with access modifier **public**, but when we discuss the other access modifiers we will change some of them accordingly. What we would like to find is how the change in the access levels of the elements (fields and methods) of the class **Dog** will be reflected, when the access is made with:

- The own body of the class **Dog**.
- The body of the class **Kid**, respectively, taking into account that **Kid** is in the same namespace (or assembly), in which the **Dog** class is defined or not.

Access Level "public"

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

Let's review both type of access to members of a class, which are matched in our classes **Dog** and **Kid**:

D	The access to the member of the class is done inside the same class **directly** (the class refers itself).
R	The access to the member of the class is done via a **reference** to an object created in the body of another class (the class refers another class).

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

Dog.cs

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

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

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

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

(D markers beside `get { return this.name; }` and `this.Bark();`)

Kid.cs

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

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

(R markers beside `Console.WriteLine("Come, " + dog.name);` and `dog.Bark();`)

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

Access Level "internal"

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

Not 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 about the access level **internal**:

Dog.cs

```
class Dog
{
   internal string name = "Doggy";

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

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

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

(Margin markers "D" appear beside `this.name` and `this.Bark()`.)

Respectively, for the class **Kid**, we discuss two cases:

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

Kid.cs

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

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

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

Kid.cs

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

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

Actually the access level **internal** for 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 other assembly to the class **Dog**, one is required to be declared **public** and in the same time its members to be declared as **public**. If the class or its members have lower visibility, the access to it from other assemblies is denied (i.e. from other Visual Studio projects which compile to different **.dll** / **.exe** file).

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

Access Level "private"

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

Therefore, if we declare the field **name** and the method **Bark()** of the class **Dog** with access modifier **private**, there is no problem to access them from the same instance of the class **Dog**, but access from any other classes is not permitted. If you try to access a private method from external class, a compilation error occur. Below is the figure about the access level **private**:

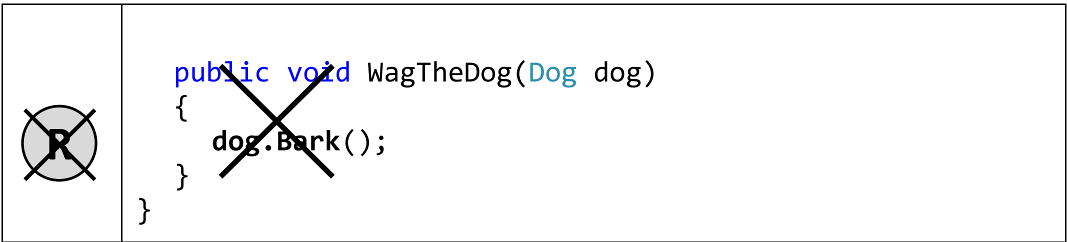
Dog.cs

```
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();
   }
}
```

Accessing the **name** fields from the same class is permitted, but accessing it from a different class (**Kid**) is restricted:

Kid.cs

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



We should know, when we assign access modifier to a filed, one in most of the cases has to be **private**, because this ensures the highest level of security applied to the field. Respectively, the access and the modification of the value from other classes (if it is required) will be done only via properties or methods. More about this technique we will learn 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 up the section regarding visibility of the elements of a class, let's try something. Let's define in the class **Dog** the field **name** and the method **Bark()** witch access modifier **private**. Let's also declare the method **Main()** with the following body:

```
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, if the class **Dog** can compile when we have declared the elements with access modifier **private** and in the same time is applied a "dot" notation to **myDog** in **Main()**?

The **compilation finished successfully**. Respectively, the result from the execution of the method **Main()** which is declared in the class **Dog** will be the following:

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

Everything works, because the access modifiers for the elements of the class are applied to the class and not to a level objects. Because the variable **myDog** is defined in the body of the class **Dog** (where also is situated **Main()** – the start method of the program), we can access its elements (fields and methods) via "dot" notation, regardless we have declared the access level as **private**. If we try to do the same in the body of the class **Kid**, this will be not possible, because the access to **private** fields from outside class 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?

Constructor of a class is a pseudo-method, which does not have a return type, has the name of the class and is **called using the keyword new**. The task of the constructor is to initialize the memory, allocated for the object, where its fields will be stored (those which are not **static** ones).

Calling a Constructor

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

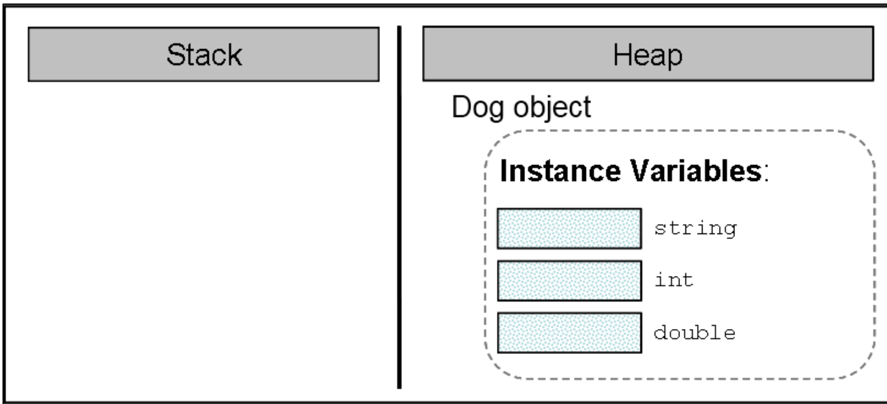
Consider an example, which will clarify how the constructor works. We know from chapter "Creating and Using Objects" how to create an object:

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

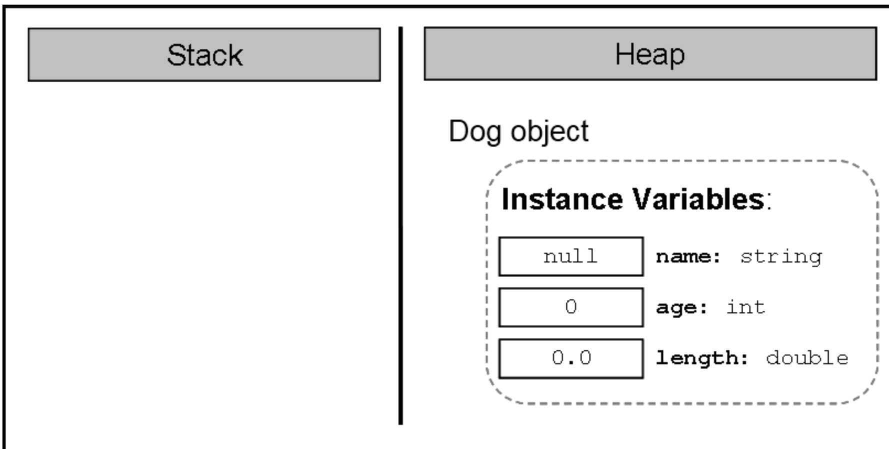
In this case by using the keyword **new** we call the constructor of the class **Dog** and by doing this, memory is allocated, needed for the newly created object of the **Dog** type. When it comes to classes they are allocated in the dynamic memory (in the so called "managed heap").

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

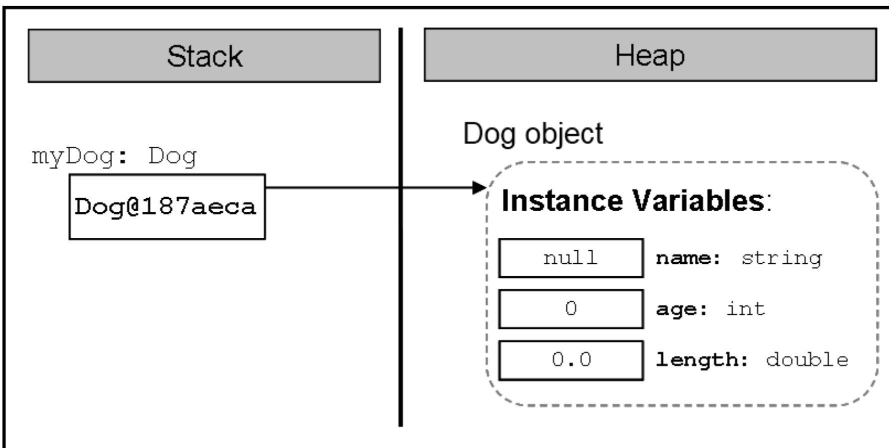
First, **memory is allocated** for the object:



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 called them pseudo-methods).

Constructor's Name

In C# it is mandatory that **the name of every constructor matches 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** (hence the name of 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 it through a **parameter list** – **<parameters_list>**. In the example constructor of the class **Dog** there is no need of 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

It is evident that **modifiers** can be added in the declaration of the constructors – **<modifiers>**. For modifiers that we know and which are not access modifiers, i.e. **const** and **static**, we should know that only **const** is not allowed to be used in 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 with fields and methods.

Initialization of the Fields in the Constructor

As explained earlier when creating a new object and calling its constructor, a new memory is allocated for the non-static fields of the object of the class and they are **initialized with the default values** for their types (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 from type **Dog** is always initialized during its declaration:

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

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

```
public class Dog
```

```
{
    private string name;

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

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

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

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

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

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

Fields Initialization in the Constructor

Let's see in details 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()
  {
  }
}
```

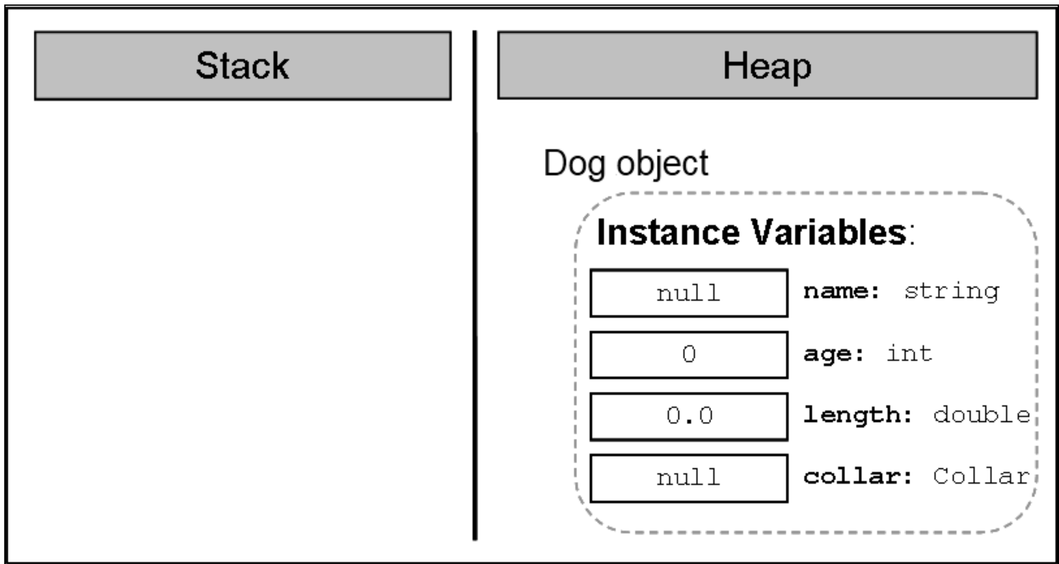
Let our class **Dog** has a field called **collar**, which is from type **Collar** and which is initialized in the constructor of the class:

```
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();
  }
}
```

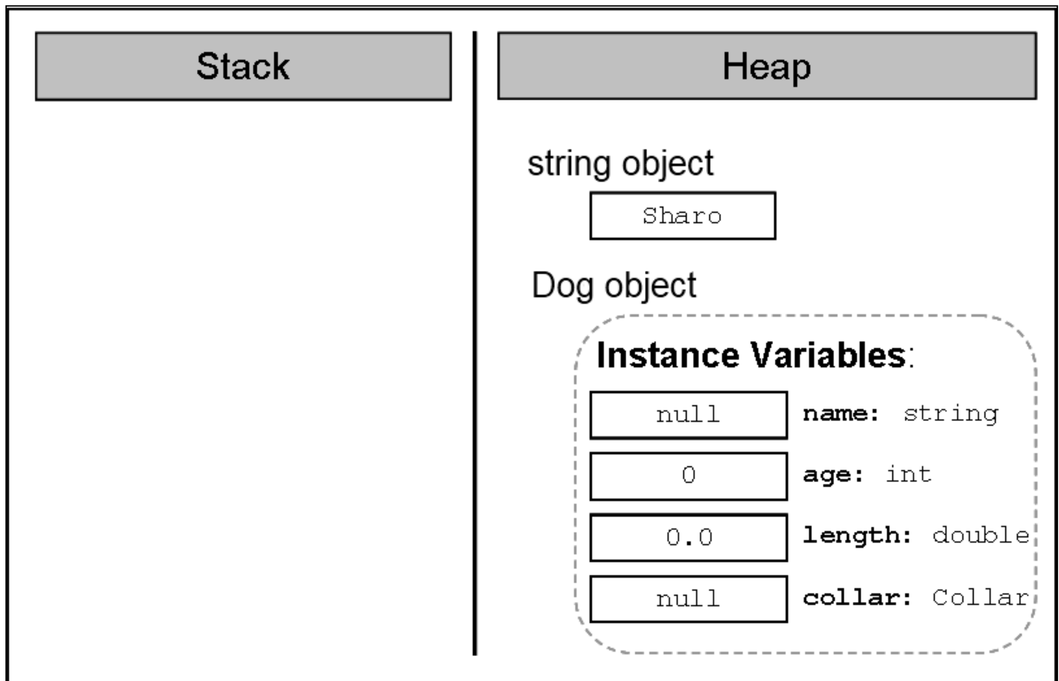
Representation in the Memory

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

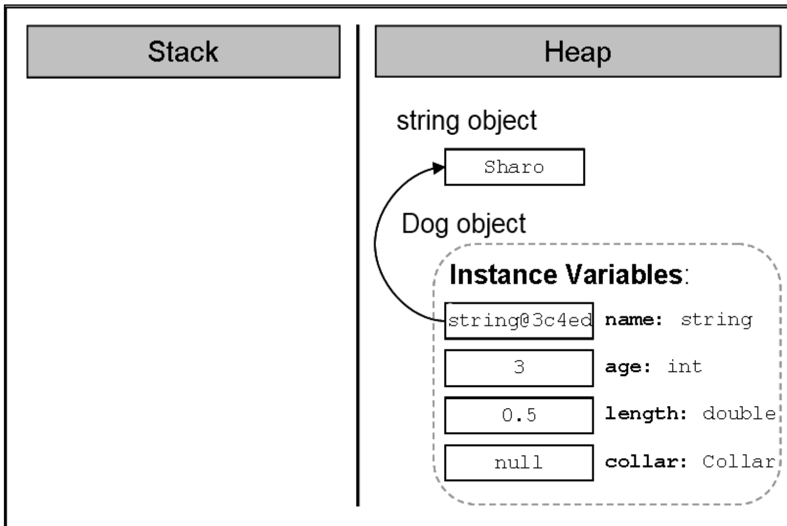
As we know, as a first step it will **allocate memory in the heap** for all the fields and will initialize them with their default values:



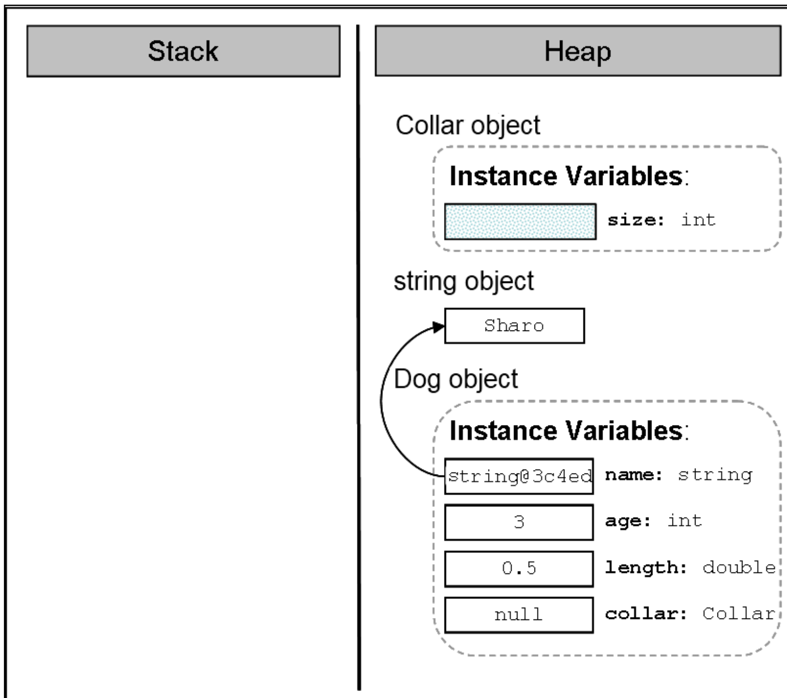
Then, the constructor will have to ensure the creation of the object for the field **name**. It will **call the constructor of the class string**, which will do the work on the string creation):



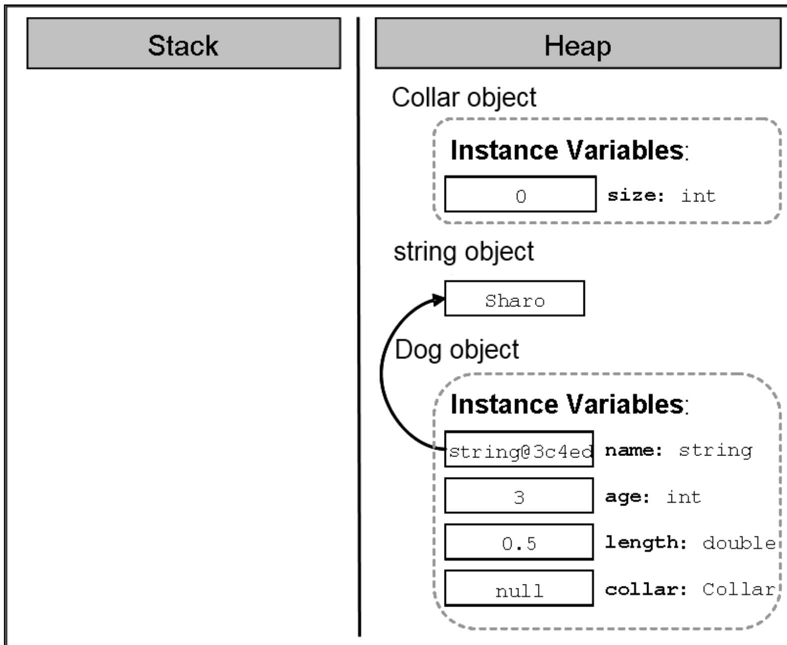
Now the constructor will keep the reference to the new string in the field **name** of the **Dog** object:



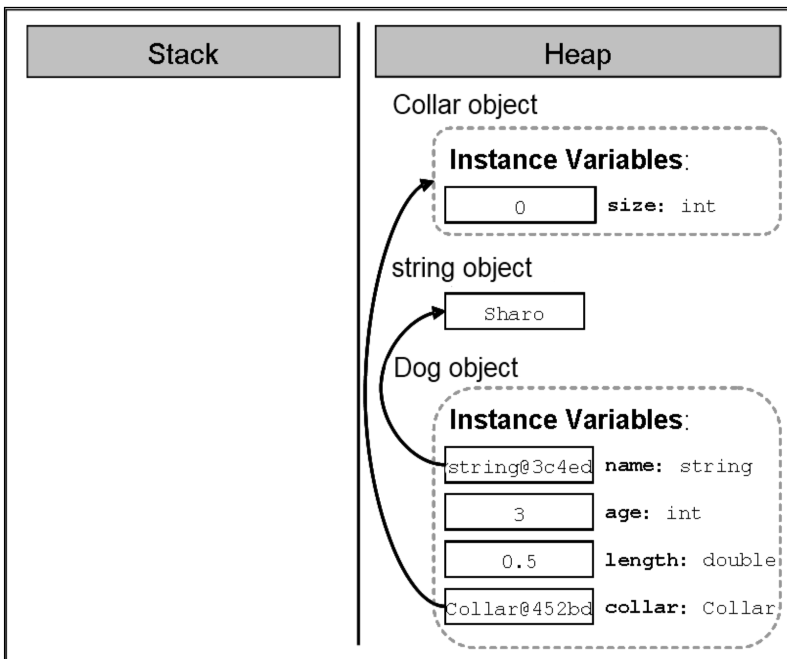
Then is the creation of the object from type **Collar**. Our constructor (of the class **Dog**) calls the constructor of the class **Collar**, which allocates memory for the object:



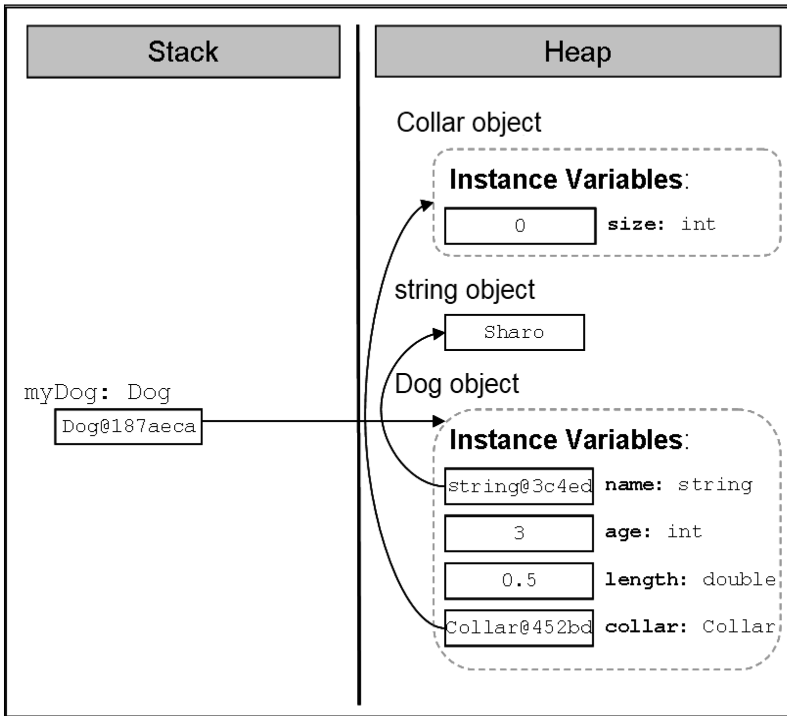
Next, the constructor will **initialize it with the default value** for the respective type. The **size** of the **Collar** is not explicitly assigned so it will take the default value for its type (**0** for **int**):



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 for the respective type, the second step in 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 of the class:

```
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 possibly 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 values to the fields, other than the default values. Very often, however, during the declaration of the constructor, we don't know what values the various fields will take. To tackle this problem, the required information, **similar to the methods with parameters**, the fields are assigned the values, given to them 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, the **call of a constructor with parameters** is done in the same way as the call of 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 the following:

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

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

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

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

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

```
age = age;
```

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

```
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 with the methods:

1. When we declare a constructor with variable number of arguments, we must use **the reserved word params**, and then insert the type of the parameters, followed by square parentheses. Finally the name of the array follows, in which array the arguments used for the calling of the

method are stored. For example for whole number arguments we can use **params int[] numbers**.

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

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

Consider a **sample 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 a sample object of this class would be constructed:

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

Accordingly, as the first parameter is the name of the subject – **"Biology"**, and all the rest arguments – the names of the attending students.

Constructor Overloading

As we saw, we can declare constructors with parameters. This gives us a possibility to declare constructors with different signatures (number and order of the parameters) with the purpose of providing convenience to those who will create objects from our class. Creating **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 = "Axl";
  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 our last example we saw that, depending on the needs for creating objects of our class, we can declare different variants of the constructors. It is easy to notice that a large part of the **constructor code is repeated**. This leads us to the question whether there is an alternative way for a constructor, which is already doing an initializing, to be reused by the others to perform the same initialization. On the other hand, at the beginning of the chapter it was mentioned that a constructor cannot be called in the manner in which the

methods are called but by the keyword **new**. There should be a way – otherwise a lot of code will be repeated unnecessarily.

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

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

To the well-known form of declaring a constructor (the first line of the declaration above), we can add a colon, followed by the keyword **this**, followed by parentheses. If the constructor we want to call has parameters, in the brackets we need to add a list of parameters **parameters_list_2** to be supplied.

Here is how the code from the section about constructor overloading would look like, in which instead of repeating the initialization of each of the fields, we will call the constructors declared in the same class:

```
// No parameters
public Dog()
   : this("Axl") // 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 indicated by comments in the first constructor in the example above, if necessary, in addition to calling any of the other constructors with certain parameters, every constructor can add into its body a code, which performs additional initializations or other actions.

Default Constructor

Consider the following question – what happens if we don't declare a constructor in our class? How can we create objects from this type?

As it often happens, when a class is without a single constructor, this issue is resolved by C#. When we do not declare any constructors, the compiler will create one for us and this one will be used to create objects such as our class. This constructor is called **default implicit constructor** and it will not have any parameters and will be empty (i.e. it will not do anything in addition to the default zeroing of the 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 in it:

```
public class Collar
{
    private int size;

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

Although we do not have an explicitly declared constructor without parameters, we can create objects of this class in the following way:

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

The **default parameterless constructor** looks the following way:

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

We should know that the default constructor is always named like the class **<class_name>**, and its parameter list is always empty as well as its body. The compiler simply adds one if there is no constructor in the class. The default constructor is usually **public** (except for some very specific situations, where it is **protected**).

The default constructor is always without parameters.

To make sure that the default constructor is always without parameters let's try to call the default constructor by setting it with parameters:

```
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?

As we can guess, the only thing the default constructor will do when creating objects of 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, and later we try to print the value in the field **size**:

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

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 a Default Constructor Will Not Be Created?

We have to know that if we declare at least one constructor in a given class then the compiler will not create a default constructor.

To investigate 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 a constructor without parameters in it, hoping that the compiler will have created a default parameterless constructor for us. 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:

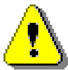
```
'Collar' does not contain a constructor that takes 0 arguments
```

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 for the constructors, we will clarify something very important:



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

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

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

Properties

In 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 in the class. In some 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 for this purpose

the programmers should declare a pair of methods (for reading and modifying the property) to provide this functionality.

Properties in C# – Introduction by Example

Using the properties is a good and proven practice and an important part of the concepts for object-oriented programming. The creation of a property in programming is done by **declaring two methods** – one for access (**reading**) and one for modifying (**setting**) the value of the respective property.

Consider an example. Assume we have again class **Dog**, which describes a dog. A characteristic of a dog is, for example, its color. The access to the property "color" of a dog and its corresponding 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** in which they are declared, i.e. to protect the class from falling into **invalid state**.

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

Though the C# syntax this is done by declaring the fields (physical presentation of data) with possibly the most limited level of visibility (mostly with the modifier **private**) and declaring that access to these fields (reading and modifying) is to take place only through special **accessor methods**.

Example of Encapsulation

To illustrate what the encapsulation, which provides properties to a class, is and what the properties themselves represent, we shall consider an example.

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

Point.cs

```
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; }
    }
}
```

The fields of the objects 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 read the properties (the coordinates) of the point only through the properties for access to them:

PointTest.cs

```
using System;

class PointTest
{
    static void Main()
    {
        Point myPoint = new Point(2, 3);

        double myPointXCoord = myPoint.X; // Access a property
        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, however, 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 it without affecting in any way of the other classes of 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 not be changed and will be the same even without changing a single character in the code of the class **PointTest**.

The demonstration is a **good example of data encapsulation** of an object, provided by the mechanism of the properties. Through them we **hide the internal representation** of the information by declaring properties and methods for accessing it, and if later a change occurs in the representation, this will not affect the other classes using our class, because they only use its properties and do not know how the information is represented "behind the scene".

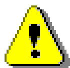
Of course, the example shows only one of the benefits of class fields wrapping (packing) into properties. **Properties allow further control over the data** in the class and they can check whether the assigned values are correct according to some criteria. For example, if we have a property "maximum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

Physical Presentation of the Properties in a Class

As we saw above, the properties may have **different presentation in one class** at a physical level. In our example, the information about the properties of the class **Point** initially was stored in two fields and later in one field–array.

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

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



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

Property without Declaration of a Field

Consider an example, in which the property is stored neither in the field, nor anywhere else, but recalculated when trying to access it.

Let's have the class **Rectangle**, which represents the geometric shape of a rectangle. Accordingly, this class has two fields – for **width** and for **height**. Assume 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 this value. Therefore, we can simply declare a method for obtaining the area through which we calculate 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 later, a property does not necessarily have an accessing and a modifying method at the same time. Therefore, it is allowed to declare only a method for reading the property **Area** of the rectangle. There is no point to have a method, which modifies the value of the area of a rectangle because the area is always one and the same based on given lengths of the sides.

Declaring Properties in C#

To declare a property in C#, we have to declare access methods (for reading and changing) of the respective property and to decide how we will store the information related to the property in the class.

Before we declare the methods, however, we have to declare the property of the class. Formal declaration of properties appears in the following way:

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

With **<modifiers>** we have denoted both the **access modifiers and other modifiers** (e.g. **static**, to be discussed in the next section of this chapter). They are not a mandatory part of the declaration of a field.

The **type of the property <property_type>** specifies the type of the values of the property. It may be either a primitive type (e.g. **int**), or a reference type (e.g. array).

Respectively, **<property_name>** is **the name of the property**. It must begin with a capital letter and to satisfy the **PascalCase** rule, i.e. every new word that is adjoined to the end part of 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 bracket "**{**" and ends with a closing bracket – "**}**". Properties should always have a body.

Method for Reading the Value of a Property (Getter)

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

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

The content of the block surrounded by the braces (**<accessor_body>**) is similar to the contents of any method. The actions, which should be performed to return the result of the method, are declared in it.

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

Although earlier in this section we considered many examples of declared properties with a method for reading their values, let's consider another example of a property – **Age**, which is of type **int** and is declared via a field in the same class:

```
private int age;             // Field declaration

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

Calling a Method for Reading Property's Value

Assume that the property **Age** from the last example is declared in the class **Dog**. Then calling the method for reading the value of the property is done by a "dot" notation, applied to a variable of the type, in the class of 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 through a dot notation the name of the property, its getter method (method for reading its value) is called automatically.

Method for Modifying Property's Value (Setter)

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

The declaration is made in the body of the property through the following syntax:

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

The contents of the block surrounded by arrow brackets – **<accessor_body>** are similar to the content of any method. It declares the actions that must be performed to change the value of the property. The method uses a hidden parameter called **value**, which is available in C# by default and contains the new value of the property. The type of the parameter is the same as the type of the property.

Let's add the example for 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

Calling the method to modify the property's value is performed via the "dot" notation, applied to the variable of the type, in the class of which the property is declared:

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

In the last line where the value 3 is assigned the setter method of the property **Age** is called. In this way the value is saved in the parameter **value** and is assigned to the setter method of the property **Age**. In our example, the value of the variable **value** is assigned to the field **age** from the class **Dog**, but in the general case this can be handled in a more complicated way.

Assertion of the Input Values

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

Consider again the example with the age of the dog. As we know the age has to be a positive number. To prevent someone from assigning a negative number or a zero 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 value to **Age**, which is a negative number or 0, the code will throw an exception from the 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 all methods, which can change a field of a class. This programming practice to protect classes from invalid data and invalid internal states is widely used and is a part of the "Defensive Programming" concept, which we will discuss in chapter "High-Quality Programming Code".

Automatic Properties in C#

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

Point.cs

```
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;
    }
}

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 code to read / write the automatically defined underlying field.

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

Types of Properties

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

1. **Read-only**, i.e. these properties have only a **get** method as shown by 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. And the most common case is **read-write**, where the property has **methods both for reading and for changing the value**.

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

Static Classes and Static Members

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 or in other words we will look at the fields, methods, properties and constructors of a class and then we will study the concept of the static class.

What the Static Elements Are Used For?

Before we study the working principle of the static elements, let's see the reasons why we need to use them.

Method to Sum Two Numbers

Let's imagine that we have 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 there is no matter exactly which object of that class is going to implement that method since it will always do the same thing – adding two numbers together, independent of the calling object.

In practice **the behavior of the method does not depend of the object state** (the values in the object field). So why the need to create an object to accomplish that method provided that the method does not depend on any of the objects of that class? Why not just get the class to implement that method?

Instance Counter for Given Class

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

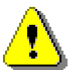
As we know, we will not be able to store the variable as a class field because for each created object there will be created a copy of that field, initialized with default value. Every single object will store its own field for indication of the number of objects and the objects will not be able to **share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

It looks like the counter should be outside a class field 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 other member, which has a **static** modifier in its declaration[1]. That means that fields, methods and properties, marked as static, belong to the particular class rather than to any particular object of the given class.

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

On the other hand if we have created objects of the given class then its **static fields and properties will be shared** and there will be only one copy of the static field or property which will be shared among all objects of the given class. Because of that reason in the VB.NET language we have the keyword **shared** instead of the **static** keyword.

Static Fields

When we create objects from a given class, each of them 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 one for the age – **age**. Every object, each of these fields, has its own value, which is stored in a different place in the memory for every object.

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

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

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

Declaration of Static Fields

The static fields are declared the same way as the class fields. If there is access modifier, 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 count of the created objects from the class **Dog**:

Dog.cs

```
public class Dog
{
    // Static (class) variable
    static int dogCount;

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

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

Initialization during Declaration

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

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

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

This initialization will complete during the first invocation to the static field. When we access some static field, an amount of memory will be reserved for it and it will be initialized with its default values. If the field has initialization during declaration (like it is in our case with the **dogCount** field) this initialization will execute. If we try later to access the field from other part of our program this process will not repeat, because the static field already exists and is initialized.

Accessing Static Fields

In contrast to the common (non-static) class fields, the static fields that are associated with the particular class can be accessed through an external class. In order to do that we need to put a **"dot" notation** this way:

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

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

```
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 method in the class, which is defined as a static field, we can access it directly without the class name, because it is known by default.

```
<static_field_name>
```

Modification of the Static Field Values

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

That's why if we want to count the number of created objects of the class **Dog**, we should use a **static field** and increment it by one every time the constructor is invoked, i.e. every time we create and object of 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 static field from the class **Dog** so we can use the following code in order to access the 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, it is clear that the field in the class **Dog** is static. The code is more readable.

Let's create some objects of the class **Dog** and print out their number in order to check if we are right:

```
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 fields.

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

In C# constants are of two types:

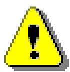
1. Constants the values of which are extracted during the compilation of the program (**compile-time constants**).
2. Constants the values of which are extracted during the execution of the program (**run-time constants**).

Compile-Time Constants (const)

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

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

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



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

For example, if we want to declare as a constant the number "PI", which is known to us from mathematics, this can be done as follows:

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

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

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

When we try to print the value of the constant:

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

The command line will display:

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

If we do not give a value to a constant at its declaration, but later, we will get a compilation error. For example, if we take the example of the constant PI, we first declare the constant and 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;
}
```

The compiler will issue an error like this one, indicating the line, where the constant is declared:

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

Let's pay attention, again:

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 consider the following example: 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 is represented by mixing the three primary colors – red, green and blue. These three primary colors are represented as three integers in the range from 0 to 255. For example, black is represented as (0, 0, 0), white as (255, 255, 255), blue – (0, 0, 255) etc.

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

Color.cs

```
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;
  }
}
```

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 their own objects for these particular colors every time. To do this, we modify the code of our class as follows, adding the declaration of the following color-constants:

Color.cs

```
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;
  }
}
```

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 be only of the following types:

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

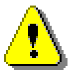
The problem with the compilation of the class in our example is connected with the reference types and the restriction on the compiler not to allow simultaneous use of the operator **new** when declaring a constant when this constant is declared with the modifier **const**, unless the reference type can be calculated at compile time.

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

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

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

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 us 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 **const** modifier.

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

Accordingly, **<reference-type>** is a type the value of which cannot be calculated at compilation time.

The compilation is successful if we replace **const** by **static readonly** in the last example of the previous section:

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

Naming the Constants

The constants names in C# follow the **PascalCase** rule according to the Microsoft's official C# coding convention. If the constant is composed of several words, each new word after the first one 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);
```

Sometimes naming in style **ALL-CAPS** is used but it is not officially supported by the Microsoft code conventions, even though it is widely distributed in programming:

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

The examples made it clear 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 common values** (literals) in the program and to enable them to change easily. The use of constants instead of brutally hardcoded repeating values facilitates readability and maintenance of the code and is 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 can make it difficult to read and maintain the code instead of making it simple. Therefore, it is believed that **values, which occur more than once in the program or are likely to change over time, must be declared as constants**.

In the chapter "High-Quality Programming Code" will we learn in details when and how to use constants efficiently.

Static Methods

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

Declaration of Static Methods

To declare a **static method** syntactically means that we must add the keyword **static** in the method's declaration:

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

Let's for example declare the method of 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) applied to the name of the class and the class name can be skipped if the calling is performed by the same class, in which the static method is declared. 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 they have been defined. For example, if we want to declare a method, which returns the number of the created objects of the **Dog** class, it must be static, because our counter will be static too:

```
public static int GetDogCount()
{
```

```
    return dogCount;
}
```

But when we examine how static and non-static methods and fields can be accessed, 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 method **PrintInfo()**, which displays information about our dog:

Dog.cs

```
public class Dog
{
    // Static variable
    static int dogCount;

    // 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.Write("wow-wow");
    }

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

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

```
    }
}
```

Of course, if we create an object of the **Dog** class and call his **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 non-static method. As we learned earlier, this is because static methods and variables are bound by class, rather than a specific method and the static elements can be accessed from any object of the class, even of external classes (as long as they are visible to them).

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 the example, we provide access to the value of the static field **PI** of the non-static method **PrintSurface()**, by calling the static method **CalculateSurface()**. Let's try to call this 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. We have declared the constant **PI**, in it. We can declare a static method for finding the length of the circle (the formula for finding perimeter of a circle is **2πr**, where **r** is the radius of the circle), that uses the constant **PI** for calculating the perimeter of a circle. Then, to show that static method can call another static method, we can call the static method for finding the perimeter of a circle 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 look at the most interesting case of a combination of accessing non-static and static elements of the class – **accessing non-static elements form a static method**.

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

Unauthorized Access to Non-Static Field – Example

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

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

Accordingly, 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 will still not be satisfied and this time will **fail to compile** the class and will display 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 non-static method from static method. Let 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);
}
```

Accordingly, let's try from the method **Main()**, which we declare in the class **Dog**, to call this method without creating an object of our class:

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

When we try to compile 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 cheat the compiler, trying to call the method via the keyword **this**:

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

Accordingly, as with the attempt to access the non-static field of a static method using the keyword **this**, the compiler displays the following 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 make 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 the class of static method has a single solution – these non-static elements are 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");
}
```

Accordingly, this code is compiled and the result is:

```
My dog "Lassie" has age of 2 years
```

Static Properties of the Class

Although rare, it is sometimes convenient to use and declare not the object characteristics, but the ones of the class. They possess the same characteristics like the properties related to the particular object of a particular class, which we discussed above, but with the difference that the **static properties refer to the class** (not its objects).

As we can guess, all we need to do to turn a simple 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. We can create many objects from it, but the model of the system has a version and a vendor, which are common to all instances created from this class. We can make the version and the vendors as static properties of the class:

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 have chosen to keep the value of static properties in static variables (which are logical, since they are bound only to the class). The properties that we consider are **Version** and **Vendor**, respectively. For each of them we have created static methods for reading and modification. Thus, all objects of this class will be able to retrieve the current version and vendor of the system, which describes the class. Accordingly, if one day an upgrade of the system version is done and the value becomes **0.2**, as a result each object will receive the new version by accessing the class 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 of 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
```

Accordingly, if we try to access the static properties through 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);
}
```

The code is compiled and 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** of the first line of the **Main()** method, by calling its method of 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 complete understanding we have to explain that we 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 as static, it is an indication that **this class contains only static members** (i.e. static fields, methods, properties) and cannot be instantiated.

The use of static classes is rare and most often associated with the **use of static methods and constants**, which do not belong to any particular object. For this reason, the details of static classes go beyond the scope of this book. Curious reader can find more information on the site of the Microsoft Developer Network (MSDN): http://msdn.microsoft.com/en-us/library/79b3xss3.aspx.

Static Constructors

To finish the section on static class members, we should mention that classes may also have **static constructor** (i.e. 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 of class is created.
2. A static element of the class is accessed (field, method, property).

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

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 these calculations to be done once at program startup and then to use the already

calculated values. Of course, to make such **pre-computing of all square roots** in a given range, we must first define this range and it should not be too wide (e.g. from 1 to 1000). Then we need, at first request for a square roots of a number, to recalculate all the square roots in this range and then to return the already calculated value. Upon a following request for a square root, all values in this range will have already been calculated and returned directly. If the program is never required to calculate the square root, preliminary calculations should not be fulfilled at all.

Through the described process initially some CPU time is invested for preliminary calculations, but then the extraction of the square root later is done very quickly. If we have multiple calculations of the square root, the pre-calculation will significantly increase the performance.

All this can be implemented in one **static class with a static constructor**, in which the square roots will be recalculated. The results, which have already been calculated, can be **stored in a static array**. A **static method** can be used to extract the already pre-calculated value. Since the preliminary calculations are being performed in the static constructor, if the class for pre-calculated square roots is not used, they will not be executed and CPU time and memory will be saved.

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" from the 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 they are **passed by value** when taken as a method parameters.

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);
    }
```

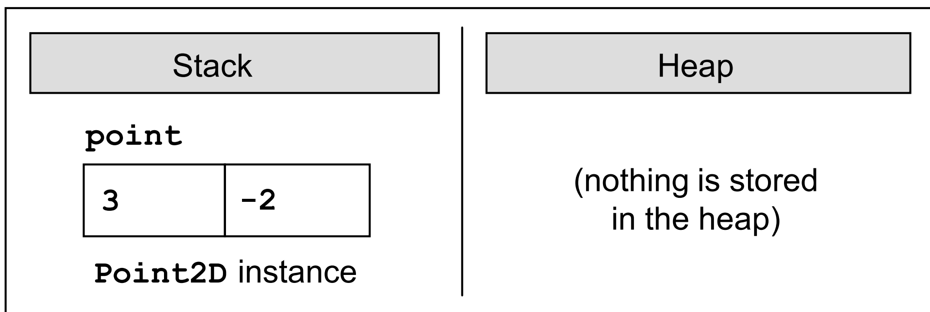
```
}
```

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

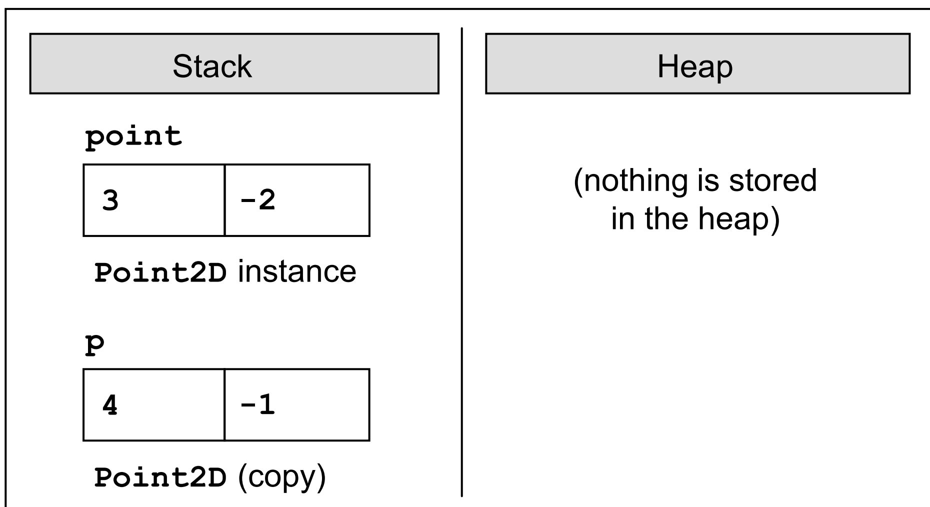
```
(3,-2)
(3,-2)
```

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):



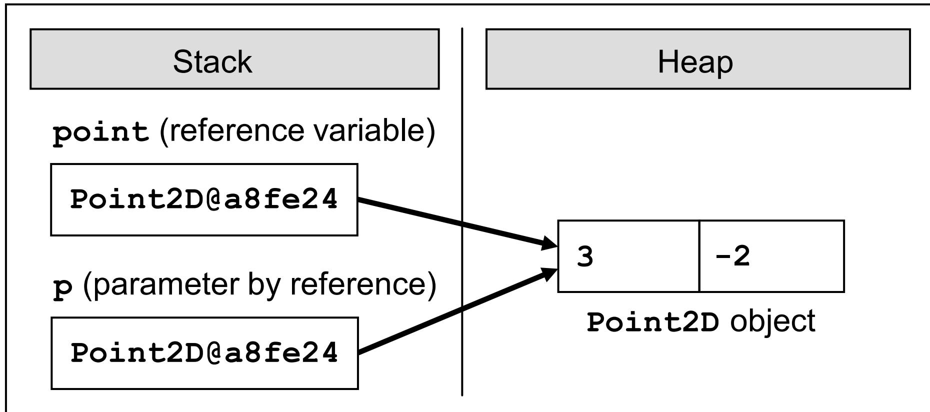
Next, the method **TryToChangePoint(Point2D p)** is called and it copies the value of the variable **point** into **another place in the stack**, allocated for the parameter **p** of the method. When the parameter **p** is changed in the method's body, it is modified in the stack and this **does not 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 come 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 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, prefer using 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 that classes are reference types and structures are values types, but we will not going to discuss them. For more details 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, how to declare and use them. In this connection 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

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

Formally speaking, the enumerations can be declared using the reserved word **enum** instead of **class**:

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

Under **<modifiers>** we understand 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 block.

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

Days.cs

```
enum Days
{
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
}
```

Naming of constants in one particular enumeration follows the same principles of naming of which we already explained in the "Naming Constants" section.

Note that each of the constants listed in the enumeration is of type this enumeration, i.e. in our case **Mon** belongs to type **Days**, as well as each of the other constants.

In other words, if we execute the following line:

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

This will be printed as a result:

```
True
```

Let's repeat again:

The enumerations are a set of constants of type – this listed type.

Nature of Enumerations

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

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

```
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 in the list of constants of this type, i.e. numbers from 0 to the number of constants in the type less 1. In this way, if we consider the example with the enumeration type for the week days, used in the previous subsection, the constant **Mon** is associated with the numerical value 0, the constant **Tue** with the integer value 1, **Wed** – with 2, etc.

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

Despite the integer nature of constants in a particular enumeration, when we try to print a particular constant, its textual representation at the time of the constant's declaration will be printed:

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

After we execute the code above we will get the following result:

```
Mon
```

Hidden Numerical Value of Constants in Enumeration

As we can guess it is possible to change the **numerical value of constants in an enumeration**. This is done when we assign the values we prefer to each of the constants at the time of declaration.

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

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

To get a clearer idea of the given definition consider the following example: let's have a class **Coffee**, which represents a cup of coffee that customers order in a coffee shop:

Coffee.cs

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

In this facility customers can order different amounts of coffee, as the coffee machine has predefined values "small" – 100 ml, "normal" – 150 ml and "double" – 300 ml. Therefore, we can declare one enumeration **CoffeeSize**, which has respectively three constants – **Small**, **Normal** and **Double**, the correspondent qualities of which will be 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 reflect 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 try to print the values of the coffee quantity for a normal and for one 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);
}
```

As we compile and execute this method, the following will be printed:

```
The Normal coffee is 150 ml.
The Double coffee is 300 ml.
```

Use of Enumerations

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

Another very important application of the enumerations is the pressure exercised by the compiler to use constants from the enumerations and not just numbers. Thus we minimize future errors in the code. For example, if we use an **int** variable instead of a variable from enumerations and a set of constants for the valid values, nothing prevents us from assigning the variable any value, e.g. -6723.

To make this clearer, consider the following example: create a class "**coffee price calculator**", which is calculating 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 capacity 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 by the submitted amount.

The problem lies in the fact that someone may decide not to use the constants defined by us and may submit an invalid number as a parameter of our method, for example: -1 or 101. In this case, if the method does not check for invalid quantity, it will likely return a wrong price, which is incorrect behavior.

To avoid this problem we will use one feature of these enumerations, namely constants in the enumeration type can be used in **switch-case** structures. 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 in previous examples:

```
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);
  }
}
```

As we can see in this example, the possibility for the users of our method to provoke unexpected behavior of the method is negligible, because we force them to use specific values which to be used as arguments, namely constants of enumerated **CoffeeSize** type. This is one of the advantages of constants, which are declared in enumeration types to constants declared in any class.

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

Before we finish with the enumeration section we should mention that the enumerations are to be used with caution when working with the **switch-case** construct. For example, if one day the owner of the coffee shop buys many big cups (mugs) for coffee, we will need to add a new constant in the constant 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 we try to calculate the coffee price with the new quantity, the method, which calculates the price, will throw an exception, informing the user that such amount of coffee is not available in the coffee shop.

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 code that already exists and uses the constants, declared so far.

Inner Classes (Nested Classes)

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

The main reason to declare one class into another are:

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

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

Declaration of Inner Classes

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

1. **`public`** – an inner class is accessible from any assembly.
2. **`internal`** – an inner class is available in the current assembly, in which is located the outer class.
3. **`private`** – access is restricted only to the class holding the inner class.
4. **`static`** – an inner class contains only static members.

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

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

internal, it needs in creating the internal class to submit a reference to an outer class.

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

Inner Classes – Example

Consider the following example:

OuterClass.cs

```
public class OuterClass
{
   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** defines into itself as a member the class **InnerClass**. Non-static inner class methods have access to their own body **this** as well as the instance of outside class **parent** (through syntax **this.parent**, if the **parent** reference is added by the developer). In the example while creating the inner class, **parent** reference is set to constructor of the outer class.

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

```
Nested name: nested
Outer name: outer
```

Usage of Inner Classes

Consider an example. Let's have a class for car – **Car.** Each car has an engine and doors. Unlike the car's door, however, the engine makes no sense regarded as being outside the car, because without it, the car cannot run, i.e. we have composition (see the section "Class Diagrams: Composition" in the chapter "Principles of Object-Oriented Programming").

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

Therefore, if you declare the class for a car: **Car** would be appropriate to create an inner class **Engine,** which will reflect the appropriate concept for the car engine:

Car.cs

```
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;
    }
}
```

Declare Enumeration in a Class

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

For example, the enumeration of type **CoffeeSize**, we have created in the previous section, can be declared inside the body of the class **Coffee**, thereby it improves its encapsulation:

Coffee.cs

```
class Coffee
{
    // Enumeration declared inside a class
    public static enum CoffeeSize
    {
        Small = 100, Normal = 150, Double = 300
    }

    // Instance variable of enumerated type
    private CoffeeSize size;

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

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

Respectively, the method for calculation of the price of coffee will be slightly modified slightly:

```
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). Before we begin, however, let's look through an example that will help us understand more easily the idea.

Shelter for Homeless Animals – Example

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

Dog.cs

```
public class Dog
{
}
```

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

Cat.cs

```
public class Cat
{
}
```

Then we want to create a class that describes a **shelter for homeless animals – AnimalShelter**. This class has a specific number of free cells, which determines the number of animals, which could find refuge in the shelter. The special feature of the class, that we want to create, is that it only needs to accommodate animals of the same kind, in our case, dogs or cats only, because the coexistence of different species is not always a good idea.

If we think about how to solve the task with the knowledge that we have until here, we will come to the 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 the universal type **object**.

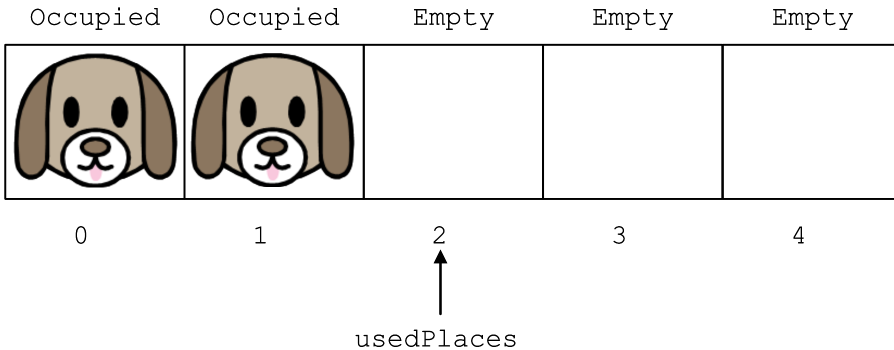
For instance, if we want to make a shelter for dogs, here is how our class would look like:

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;
    }
}
```

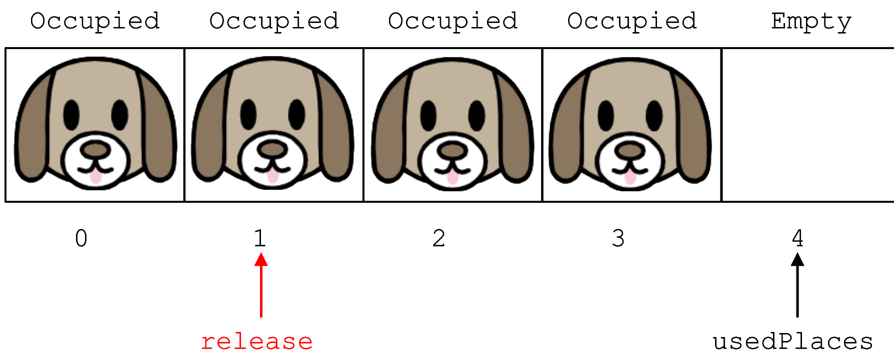
Shelter capacity (number of animals, which it is capable to accommodate) is set when the object is created. By default it is the value of the constant **DefaultPlacesCount**. We use the field **usedPlaces** to monitor the occupied cells (at the same time we use it to index into the array for "pointing" to the first space from left to right in the array).



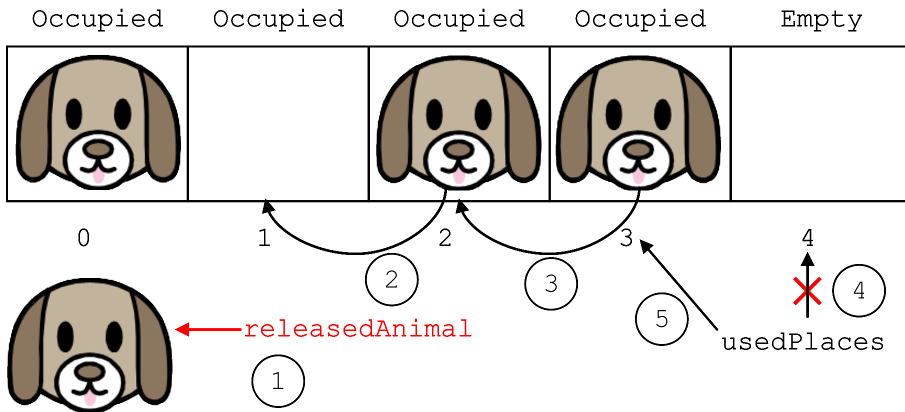
We have created a method for adding a new dog into the shelter – **Shelter()** and respectively for releasing from the shelter – **Release(int)**.

The method **Shelter()** adds each new animal in the first free cell in the right side of the array (if there is any free).

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

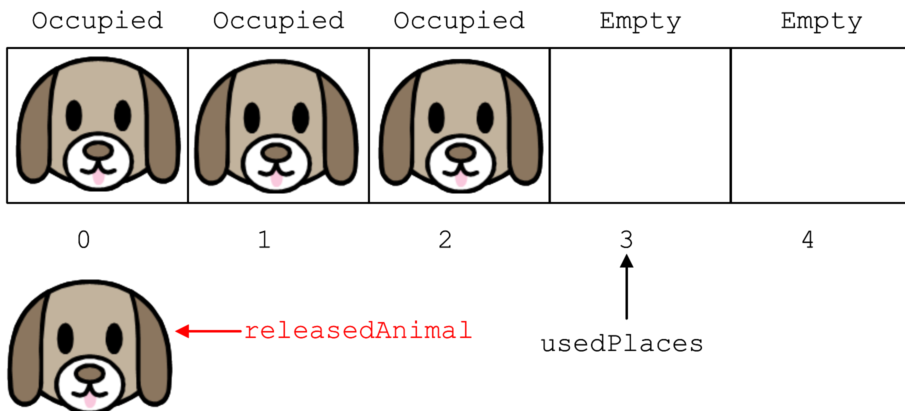


Then it moves all animals which are having a bigger cell number then the current cell, from which we will release a dog, with a position to the left (steps 2 and 3 are shown in the diagram below).



Released cell at position **usedPlaces-1** is marked as free, and a value **null** is assigned to it. This provides release of the reference to it and respectively allows the system to clean memory (garbage collector), to release the object if it is not used anywhere else in the program at this moment. This prevents from indirect loss of memory (memory leak).

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



It is visible that the "removal" of an animal from a cell **could be a slow operation**, because it requires the transfer of all animals from the next cells with one position left. In the chapter "Linear Data Structures" we will discuss also more efficient ways of presenting the animal shelter, but for now let's focus on the topic about generic types.

So far we succeed implementing functionality of the 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 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 **AnimalShelter** class and change only the type of the objects, which are handled – **Cat**.

Yes, but if we decide to make a shelter for other species? How many classes of shelters for the particular type of animals we shall create?

We can see that this solution of the problem **is not sufficiently comprehensive** and does not fully meets the terms, which we were set – to exist a **single class** that describes our shelter **for any kind of animal** (i.e. for all objects) and by working with it, **it should contain only one kind of animals** (i.e. only objects of one and the same type).

We could use instead of the type **Dog**, the universal type **object**, which can take values as **Dog**, **Cat** and all other data types, but this will create some inconvenience, associated with the need to convert back from the **object** to the **Dog**, when creating a shelter for dogs and it 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. It is called **generics** (template classes).

What Is a Generic Class?

As we know if 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.

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

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

To get a cleaner picture of the nature of the generic types, let's return to our task from the previous section. As you might guess, the class that describes the animal shelter (**AnimalShelter**) **can operate with different types of animals**. Consequently, if we want to create a general solution of the task, during the declaration of class **AnimalShelter**, we cannot know what type of animals will be sheltered to shelter. This is sufficient indication, that we can typify our class, adding to the declaration of the class as a parameter, the unknown type of 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.

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

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, which will be used later:

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

It should be noticed that the characters '**<**' and '**>**', which surround the substitution **T** are an obligatory part of the syntax of language C# and must participate in the declaration of a generic class.

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

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

Let's can imagine that we are creating a **template of our class AnimalShelter**, which we will specify later, replacing **T** with a specific type, for instance a **Dog**.

A particular class may have more than one substitute (to be parameterized by more than one type), depending on its needs:

```
[<modifiers>] class <class_name><T1 [, T2, [… [, Tn]]]>
{
}
```

If the class needs **several different unknown types**, these types should be listed by a comma between the characters '**<**' and '**>**' in the declaration of the class, as each of the substitutes used must be different identifier (e.g. a different letter) – in the definition they are indicated as **T1**, **T2**, …, **Tn**.

In case, we should to create a shelter for animals of a mixed type, one that accommodates both – dogs and cats, we should declare the class as follows:

```
class AnimalShelter<T, U>
{
  // Class body here …
}
```

If this were our case, we would use the first parameter **T**, to indicate objects of type **Dog**, which our class would operate with, and with **U** – 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.** The using of generic classes should be done as follows:

```
<class_name><concrete_type><variable_name> =
  new <class_name><concrete_type>();
```

Again, similar to **T** substitution in the declaration of our class, the characters '**<**' and '**>**' surrounding a particular class **concrete_type**, are required.

If we want to 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 a 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 parameters that are used to indicate the unknown types are visible in the whole body of the class, therefore they can be used to declare the field as each other type:

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

As we can guess, in our example with shelter for homeless animals, we can use this feature provided by language C#, to declare the type of field **animalsList**, which holds references to objects for the housed animals, instead of a specific type of **Dog**, with parameter **T**:

```
private T[] animalList;
```

Let's assume when we create an object of our class, setting a specific type (e.g. **Dog**) during the execution of the program, t**he unknown type T will be replaced** with the above type. If we choose to create a shelter for dogs, we can consider that our field is 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 – creating an array, using substitution of the unknown type – **T**:

```
public AnimalShelter(int placesNumber)
{
   animalList = new T[placesNumber]; // Initialization
   usedPlaces = 0;
}
```

Using Unknown Types in a Method's Declaration

As an **unknown type** used in the declaration of a generic class is visible from opening to closing brace of the class body, except for field's declaration, it **can be used in a method declaration**, namely:

As a parameter in the list of parameters of the method:

```
<return_type> MethodWithParamsOfT(T param)
```

- As a result of implementation of the method:

```
T MethodWithReturnTypeOfT(<params>)
```

As we already guessed, using our example, we can adapt the methods **Shelter(…)** and **Release(…)**, respectively:

- As a method of unknown type parameter **T**:

```
public void Shelter(T newAnimal)
{
   // Method's body goes here …
}
```

- And a method, which returns a result of unknown type **T**:

```
public T Release(int i)
{
   // Method's body goes here …
}
```

As we already know when we create an object from our class shelter and replace the unknown type with a specific one (e.g. **Cat**), during the execution of the program, the above methods will have the following form:

- The parameter of method **Shelter** will be of type **Cat**:

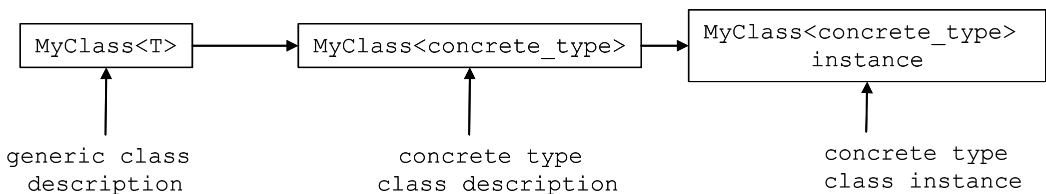
```
public void Shelter(Cat newAnimal)
{
  // Method's body goes here …
}
```

- The method **Release** will return a result of type **Cat**:

```
public Cat Release(int i)
{
  // Method's body goes here …
}
```

Typifying (Generics) – Behind the Scenes

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



First we declare our generic class **MyClass<T>** (generic class description in the scheme above). Then the compiler translates our code to an intermediate language (MSIL), as translated code contains information that the class is generic, i.e. it works with undefined types until now. At runtime, when someone tries to work with our generic class and tries to use it with a specific type, a new **description of the class** is created (specific type class description in the diagram above), which is identical to the generic class, with the difference that where it has been used **T**, now is replaced by a specific type. For example, if you try to use **MyClass<int>**, everywhere in your code, where the unknown parameter **T** is used, it will be replaced with **int**. Only then we can create object of a generic class with a specific type **int**. The interesting thing here is that to create this object, the description of the class, which was created in the meantime (specific type class description), will be used. Instantiating of a generic class by given specific types of its parameters is called "**specialization of the type**" or "**extension of generic class**".

Using our example, if we create an object of type **AnimalShelter<T>**, which works only with objects of type **Dog**, if we try to add an object of type **Cat**, this will cause a compile error almost identical to the errors, that were derived by an attempt to add an object of type **Cat**, into an object of type **AnimalShelter**, which we have created in 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 method's parameters cannot be specified, we can **parameterize (typify) the method**. Accordingly, the indication of a specific type will happen during the invocation of the method, replacing the unknown type with a specific one, as we did in the classes.

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

```
<return_type><methods_name><K>(<params>)
```

Accordingly, we can use unknown type **K** for parameters in the parameter's list of method **<params>**, whose type is unknown and also for return value or to declare variables of type substitute **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 carrying of their types**. That is why we define it as a generic, so we can use it for all types of variables.

Accordingly, if we want to swap the values of two integers and then two string variables, we should use our method:

```
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 as expected:

```
Before swap: 3 5
After swap: 5 3

Before swap: Hello There!
After swap: There Hello!
```

We notice that in the list of parameters we have used also the keyword **ref**. This concerns the specification of the method – namely, to exchange the values of two references. By using the keyword **ref**, the method will use the same reference that was given by the calling method. This way, all changes on this variable made by our method, will remain after the method exits.

We should know that by **calling a generic method**, we can miss the explicit declaration of a specific type (in our example **<int>**), because the compiler will detect it automatically, recognizing the type of the given parameters. In other words, our code can be simplified using the following calls:

```
Swap(ref num1, ref num2); // Invoking the method Swap<int>
Swap(ref str1, ref str2); // Invoking the method Swap<string>
```

We should know that the **compiler will be able to recognize what is the specific type**, only if this type is involved in the parameter's list. The compiler cannot recognize what is the specific type of a generic method only by the type its return value or if it does not have parameters. In both cases, this specific type will have to be given explicitly. In our example, it will be similar to the original method call, or by adding **<int>** or **<string>**.

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

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

Features by Declaration of Generic Methods in Generic Classes

As we have already seen in the section "Using Unknown Types in a Declaration of Methods", non-generic methods can use unknown types, described 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 is used to mark the unknown type of the generic class, for example as **T**, in the declaration of generic method, then when we try to compile the class, we will get a warning **CS0693**. This is happening because the scope of action of the unknown type **T**, defined in declaration of the method, overlaps the scope of action of the unknown type **T**, in class declaration:

CommonOperations.cs

```
public class CommonOperations<T>
{
   // CS0693
   public void Swap<T>(ref T a, ref T b)
   {
```

```
        T oldA = a;
        a = b;
        b = oldA;
    }
}
```

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 to be called with a specific type, different from that in the generic class by instantiating it, we just have to declare the replacement of the unknown type in the declaration of the generic method **to be different than the parameter for the unknown type** in the class declaration, as shown below:

CommonOperations.cs

```
public class CommonOperations<T>
{
    // No warning
    public void Swap<K>(ref K a, ref K b)
    {
        K oldA = a;
        a = b;
        b = oldA;
    }
}
```

Thus, always make sure that there will be no overlapping of substitutes of the unknown types of method and class.

Using a Keyword "default" in a Generic Source Code

Once we have introduced the basics of generic types, let's try to **redesign our first example** in this section (Shelter for Homeless Animals). The only thing we need to do is to replace the type **Dog** with some 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 looks to work properly, until 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 is inside the method **Release()** and it is related to the recording a **null** value in the last released (rightmost) cell in 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 errors above. 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, in our code we have to use the construct **default(T)** instead of **null**, which returns the default value for the particular type that will be used instead of **T**. As we know, the default value for reference type is **null**, and for numeric types – zero. We can make the following change:

```
// this.animalList[this.usedPlaces - 1] = null;
this.animalList[this.usedPlaces - 1] = default(T);
```

Finally the compilation runs smoothly and the class **AnimalShelter<T>** operates correctly. We can test it 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**, the security and the performance compared to other non-generic alternatives.

As a general rule, the **programmer should strive to create and use generic classes, whenever it is possible**. The more generic types are used, the higher level of abstraction there is in the program and the source code becomes more flexible and reusable. 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 finish generics as a topic, let's give you some guidance on working with the substitutes (parameters) of unknown types in a generic class:

1. If there is just one unknown type in the generic, it is common to use the letter **T**, as a substitute for that unknown type. As an example we can give our class declaration **AnimalShelter<T>**, which we used until now.

2. To the substitutes should be given the most descriptive names, unless a letter is not a sufficiently descriptive and well-chosen name, this will not improve readability of the source code. For instance, we can modify our example, replacing the letter **T**, with the more descriptive substitute for **Animal**:

AnimalShelter.cs

```
public class AnimalShelter<Animal>
{
    // … The rest of the code …

    public void Shelter(Animal newAnimal)
    {
        // Method body here
    }

    public Animal Release(int i)
    {
        // Method body here
    }
}
```

When we use descriptive names of substitutes instead of a letter, it is better to add **T** at the beginning of the name, to distinguish it more easily from the class names in our application. In other words, instead of using a substitute **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 to be initialized with **null**. Use nullable types for all non-mandatory data.

3. Add a **static field** for the class **Student**, which holds the number of created objects of this class.
4. Add a **method** in the class **Student**, which displays complete information about the student.
5. Modify the current source code of **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** in class **StudentTest**, which creates several objects of type **Student** and store them in static fields. Create a **static property** of the class to access them. Write a test program, which displays the information about them 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 by the previous task, 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 method, which deletes all calls from the archive.

18. In **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 school we have **classes** and **students**. Each class has a number of **teachers**. Each teacher has a variety of disciplines taught. Students have a name and a unique number in the class. Classes have a unique text identifier. Disciplines have a name, number of lessons and number of exercises. The task is to shape a school with C# classes. You have to define 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 of 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** of the array from the previous task, when by adding an element, it reaches the capacity of the array.

25. Define a class **Fraction**, which contains information about the **rational fraction** (e.g. ¼ or ½). Define a static method **Parse()** to create a fraction from a sting (for example **-3/4**). Define the appropriate

properties and constructors of the class. Also write 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 previous task. Pay close attention on testing the function Parse with different input data.
27. Write a function to **cancel a fraction** (e.g. if numerator and denominator are respectively 10 and 15, fraction to 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 as a place where the number of objects of class **Student** is increasing.
4. Display on the console in all fields of the class **Student**, followed by a blank line.
5. Define as **private** all members of the class **Student** and then using Visual Studio (Refactor -> Encapsulate Field) define automatically the public **get** / **set** methods to access these fields.
6. **Create a few students** and display the whole information for each one of them.
7. You can use the **static constructor** to create instances in the 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, except 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()** in 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 into 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 of their greatest common divisor and you will get the cancelled fraction.