

Chapter 20. Object-Oriented Programming Principles

In This Chapter

In this chapter we will familiarize ourselves with the **principles of object-oriented programming**: class **inheritance**, **interface** implementation, **abstraction** of data and behavior, **encapsulation** of data and class implementation, **polymorphism** and **virtual methods**. We will explain in details the principles of **cohesion** and **coupling**. We will briefly outline object-oriented modeling and how to create an object model based on a specific business problem. We will familiarize ourselves with **UML** and its role in object-oriented modeling. Finally, we will briefly discuss **design patterns** and illustrate some of those that are widely used in practice.

Let's Review: Classes and Objects

We introduced classes and objects in the chapter "[Creating and Using Objects](#)". Let's shortly review them again.

Classes are a description (**model**) of real objects and events referred to as entities. An example would be a class called "Student".

Classes possess **characteristics** – in programming they are referred to as **properties**. An example would be a set of grades.

Classes also expose **behavior** known in programming as **methods**. An example would be sitting an exam.

Methods and properties can be **visible** only within the scope of the class, which declared them and their descendants (**private / protected**), or visible to all other classes (**public**).

Objects are **instances** of classes. For example, John is a Student and Peter is also a Student.

Object-Oriented Programming (OOP)

Object-oriented programming is the successor of procedural (structural) programming. **Procedural programming** describes programs as groups of reusable code units (procedures) which define input and output parameters. Procedural programs consist of **procedures**, which invoke each other.

The **problem** with procedural programming is that **code reusability is hard** and limited – only procedures can be reused and it is hard to make them generic and flexible. There is no easy way to work with abstract data structures with different implementations.

The object-oriented approach relies on the paradigm that each and every program works with data that describes **entities** (objects or events) from real life. For example: accounting software systems work with invoices, items, warehouses, availabilities, sale orders, etc.

This is how objects came to be. They describe characteristics (properties) and behavior (methods) of such **real life entities**.

The main advantages and goals of OOP are to make complex software faster to develop and easier to maintain. OOP enables the easy reuse of code by applying simple and widely accepted rules (principles). Let's check them out.

Fundamental Principles of OOP

In order for a programming language to be **object-oriented**, it has to enable working with **classes** and **objects** as well as the implementation and use of the fundamental object-oriented principles and concepts: inheritance, abstraction, encapsulation and polymorphism. Let's summarize each of these **fundamental principles of OOP**:

- **Encapsulation**

We will learn to **hide unnecessary details** in our classes and provide a clear and simple interface for working with them.

- **Inheritance**

We will explain how **class hierarchies** improve code readability and enable the reuse of functionality.

- **Abstraction**

We will learn how to **work through abstractions**: to deal with objects considering their important characteristics and ignore all other details.

- **Polymorphism**

We will explain how to work in the same manner with different objects, which define a specific implementation of some **abstract behavior**.

Some OOP theorists also put the concept of **exception handling** as additional **fifth fundamental principle of OOP**. We shall not get into a detailed dispute about whether or not exceptions are part of OOP and rather will note that **exceptions are supported in all modern object-oriented languages** and are the primary mechanism of handling errors and unusual situations in object-oriented programming. **Exceptions always come together with OOP** and their importance is explained in details in the chapter "[Exception Handling](#)".

Inheritance

Inheritance is a fundamental principle of object-oriented programming. It allows a class to "inherit" (behavior or characteristics) of another, more general class. For example, a lion belongs to the biological family of cats (Felidae). All cats that have four paws, are predators and hunt their prey. This functionality can be coded once in the **Felidae** class and all its predators can reuse it – **Tiger**, **Puma**, **Bobcat**, etc. Inheritance is described as **is-kind-of relationship**, e.g. **Tiger** is kind of **Animal**.

How Does Inheritance Work in .NET?

Inheritance in .NET is defined with a special construct in the class declaration. In .NET and other modern programming languages, a class can inherit from a single class only (**single inheritance**), unlike C++ which supports inheriting from multiple classes (**multiple inheritance**). This limitation is necessitated by the difficulty in deciding which method to use when there are duplicate methods across classes (in C++, this problem is solved in a very complicated manner). In .NET, classes can inherit multiple interfaces, which we will discuss later.

The class from which we inherit is referred to as **parent class** or **base class** / **super class**.

Inheritance of Classes – Example

Let's take a look at an example of class inheritance in .NET. This is how a base class looks like:

Felidae.cs

```

/// <summary>Felidae is latin for "cats"</summary>
public class Felidae
{
    private bool male;

    // This constructor calls another constructor
    public Felidae() : this(true)
    {}

    // This is the constructor that is inherited
    public Felidae(bool male)
    {
        this.male = male;
    }

    public bool Male
    {

```

```
    get { return male; }
    set { this.male = value; }
}
}
```

This is how the inheriting class, **Lion**, looks like:

Lion.cs

```
public class Lion : Felidae
{
    private int weight;

    // Keyword "base" will be explained in the next paragraph
    public Lion(bool male, int weight) : base(male)
    {
        this.weight = weight;
    }

    public int Weight
    {
        get { return weight; }
        set { this.weight = value; }
    }
}
```

The "base" Keyword

In the above example, we used the **keyword base** in the constructor of the class **Lion**. The keyword indicates that **the base class** must be used and allows access to its methods, constructors and member variables. Using **base()**, we can call the constructor of the base class. Using **base.Method(...)** we can invoke a method of the base class, pass parameters to it and use its results. Using **base.field** we can get the value of a member variable from the base class or assign a different one to it.

In .NET, methods inherited from the base class and declared as **virtual** can be **overridden**. This means **changing their implementation**; the original source code from the base class is ignored and new code takes its place. More on overriding methods we will discuss in "[Virtual Methods](#)".

We can invoke non-overridden methods from the base class without using the keyword **base**. Using the keyword is required only if we have an overridden method or variable with the same name in the inheriting class.



The keyword `base` can be used explicitly for clarity. `base.method(...)` calls a method, which is necessarily from the base class. Such source code is easier to read, because we know where to look for the method in question.

Bear in mind that using the keyword `this` is not the same. It can mean accessing a method from the current, as well as the base class.

You can take a look at the example in the section about [access modifiers and inheritance](#). There it is clearly explained which members of the base class (methods, constructors and member variables) are **accessible**.

Constructors with Inheritance

When inheriting a class, our constructors must call the base class constructor, so that it can **initialize its member variables**. If we do not do this explicitly, the compiler will place a call to the parameterless base class constructor, `base()`, at the beginning of all our inheriting class' constructors. Here is an example:

```
public class ExtendingClass : BaseClass
{
    public ExtendingClass() { ... }
}
```

This actually looks like this (spot the differences):

```
public class ExtendingClass : BaseClass
{
    public ExtendingClass() : base() { ... }
}
```

If the base class has no default constructor (one without parameters) or that constructor is hidden, our constructors need to explicitly call one of the other base class constructors. The omission of such a call will result in a compile-time error.



If a class has private constructors only, then it cannot be inherited.

If a class has private constructors only, then this could indicate many other things. For example, no-one (other than that class itself) can create instances of such a class. Actually, that's how one of the most popular design patterns (Singleton) is implemented.

The Singleton design pattern is described in details at [the end of this chapter](#).

Constructors and the Keyword "base" – Example

Take a look at the `Lion` class from our last example. It does not have a default constructor. Let's examine a class inheriting from `Lion`:

AfricanLion.cs
<pre> public class AfricanLion : Lion { // ... // If we comment out the ": base(male, weight)" line // the class will not compile. Try it. public AfricanLion(bool male, int weight) : base(male, weight) {} public override string ToString() { return string.Format("(AfricanLion, male: {0}, weight: {1})", this.Male, this.Weight); } // ... } </pre>

If we comment out the line `":base(male, weight);"`, the class `AfricanLion` will not compile. Try it.



Calling the constructor of a base class happens outside the body of the constructor. The idea is that the fields of the base class should be initialized before we start initializing fields of the inheriting class, because they might depend on a base class field.

Access Modifiers of Class Members and Inheritance

Let's review: in the "[Defining Classes](#)" chapter, we examined the basic **access modifiers**. Regarding members of a class (methods, properties and member variables) we examined the modifiers **public**, **private** and **internal**. Actually, there are two other modifiers: **protected** and **protected internal**. This is what they mean:

- **protected** defines class members which are not visible to users of the class (those who initialize and use it), but are **visible to all inheriting classes** (descendants).

- **protected internal** defines class members which are both **internal**, i.e. visible within the entire assembly, and **protected**, i.e. not visible outside the assembly, but visible to classes who inherit it (even outside the assembly).

When a base class is inherited:

- All of its **public**, **protected** and **protected internal** members (methods, properties, etc.) are **visible** to the inheriting class.
- All of its **private** methods, properties and member-variables are **not visible** to the inheriting class.
- All of its **internal** members are visible to the inheriting class, only if the base class and the inheriting class are **in the same assembly** (the same Visual Studio project).

Here is an example, which demonstrates the **levels of visibility** with inheritance:

Felidae.cs
<pre> /// <summary>Latin for "cats"</summary> public class Felidae { private bool male; public Felidae() : this(true) {} public Felidae(bool male) { this.male = male; } public bool Male { get { return male; } set { this.male = value; } } } </pre>

And this is how the class **Lion** looks like:

Lion.cs
<pre> public class Lion : Felidae { private int weight; } </pre>

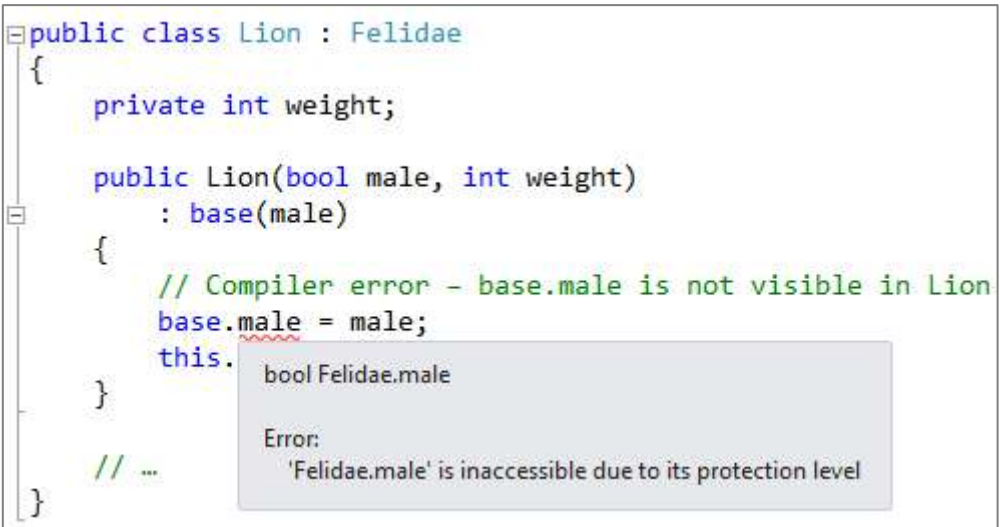
```

public Lion(bool male, int weight)
    : base(male)
{
    // Compiler error - base.male is not visible in Lion
    base.male = male;
    this.weight = weight;
}

// ...
}

```

If we try to compile this example, we will get an **error message**, because the **private** variable **male** in the class **Felidae** is not accessible to the class **Lion**:



```

public class Lion : Felidae
{
    private int weight;

    public Lion(bool male, int weight)
        : base(male)
    {
        // Compiler error - base.male is not visible in Lion
        base.male = male;
        this.
    }

    // ...
}

```

bool Felidae.male

Error:
'Felidae.male' is inaccessible due to its protection level

The System.Object Class

Object-oriented programming practically became popular with **C++**. In this language, it often becomes necessary to code classes, which must work with objects of any type. C++ solves this problem in a way that is not considered strictly object-oriented (by using **void** pointers).

The architects of .NET take a different approach. They create a class, which **all other classes inherit** (directly or indirectly). All objects can be perceived as instances of this class. It is convenient that this class contains important methods and their default implementation. This class is called **Object** (which is the same as **object** and **System.Object**).

In .NET **every class**, which does not inherit a class explicitly, **inherits the system class System.Object** by default. The compiler takes care of that.

Every class, which inherits from another class indirectly, inherits **Object** from it. This way every class inherits explicitly or implicitly from **Object** and contains all of its fields and methods.

Because of this property, **every class instance can be cast to Object**. A typical example of the advantages of implicit inheritance is its use with data structures, which we saw in the chapters on data structures. Untyped list structures (like **System.Collections.ArrayList**) can hold all kinds of objects, because they treat them as instances of the class **Object**.



The generic types (generics) have been provided specifically for working with collections and objects of different types (generics are further discussed in the chapter "[Defining Classes](#)"). They allow creating typified classes, e.g. a collection which works only with objects of type Lion.

.NET, Standard Libraries and Object

In .NET, there are a lot of predefined classes (we already covered a lot of them in the chapters on [collections](#), [text files](#) and [strings](#)). These classes are part of the .NET framework; they are available wherever .NET is supported. These classes are referred to as **Common Type System (CTS)**.

.NET is one of the first frameworks, which provide such an extensive set of predefined classes. A lot of them work with **Object** so that they can be used in as many situations as possible.

.NET also provides a lot of libraries, which can be referenced additionally, and it stands to reason that they are called class libraries or external libraries.

The Base Type Object Upcasting and Downcasting – Example

Let's take a closer look at the **Object class** using an example:

ObjectExample.cs

```
public class ObjectExample
{
    static void Main()
    {
        AfricanLion africanLion = new AfricanLion(true, 80);
        // Implicit casting
        object obj = africanLion;
    }
}
```

In this example, we cast an **AfricanLion** to **Object**. This operation is called **upcasting** and is permitted because **AfricanLion** is an indirect child of the **Object** class.



Now it is the time to mention that the keywords `string` and `object` are simply compiler tricks and are substituted with `System.String` and `System.Object` during compilation.

Let's continue with the example:

ObjectExample.cs

```
// ...  
  
AfricanLion africanLion = new AfricanLion(true, 80);  
// Implicit casting  
object obj = africanLion;  
  
try  
{  
    // Explicit casting  
    AfricanLion castedLion = (AfricanLion) obj;  
}  
catch (InvalidCastException ice)  
{  
    Console.WriteLine("obj cannot be downcasted to AfricanLion");  
}
```

In this example, we **cast an Object to AfricanLion**. This operation is called **downcasting** and is permitted only if we indicate the type we want to cast to, because **Object** is a parent class of **AfricanLion** and it is not clear if the variable **obj** is of type **AfricanLion**. If it is not, an **InvalidCastException** will be thrown.

The Object.ToString() Method

One of the most commonly used methods, originating from the class **Object** is **ToString()**. It returns a **textual representation of an object**. Every object includes this method and therefore has a textual representation. This method is used when we print the object using **Console.WriteLine(...)**.

Object.ToString() – Example

Here is an example in which we call the **ToString()** method:

TostringExample.cs

```
public class ToStringExample  
{  
    static void Main()  
}
```

```

{
    Console.WriteLine(new object());
    Console.WriteLine(new Felidae(true));
    Console.WriteLine(new Lion(true, 80));
}

```

The result is:

```

System.Object
Chapter_20_OOP.Felidae
Chapter_20_OOP.Lion

```

In this case, the base class implementation is called, because **Lion doesn't override ToString()**. **Felidae** also doesn't override the method; therefore, we actually call the implementation **inherited from System.Object**. The result above contains the namespace of the object and the name of the class.

Overriding ToString() – Example

We will now demonstrate how useful overriding **ToString()** inherited from **System.Object** can be:

AfricanLion.cs

```

public class AfricanLion : Lion
{
    // ...

    public override string ToString()
    {
        return string.Format(
            "(AfricanLion, male: {0}, weight: {1})",
            this.Male, this.Weight);
    }

    // ...
}

```

In the source code above, we use the method **String.Format(...)**, in order to format the result appropriately. This is how we can then invoke the overridden method **ToString()**:

OverrideExample.cs

```

public class OverrideExample

```

```

{
    static void Main()
    {
        Console.WriteLine(new object());
        Console.WriteLine(new Felidae(true));
        Console.WriteLine(new Lion(true, 80));
        Console.WriteLine(new AfricanLion(true, 80));
    }
}

```

The result is:

```

System.Object
Chapter_20_OOP.Felidae
Chapter_20_OOP.Lion
(AfricanLion, male: True, weight: 80)

```

Notice that `ToString()` is invoked implicitly. When we pass an object to the `WriteLine()` method, that object provides its string representation using `ToString()` and only then it is printed to the output stream. That way, there's no need to explicitly get string representations of objects when printing them.

Virtual Methods: the "override" and "new" Keywords

We need to explicitly instruct the compiler that we want our method to **override** another. In order to do this, we use the **override keyword**. Notice what happens if we remove it:

```

public string ToString()
{
    // ...
    this.male, this.weight);
}

```

Let's experiment and use the **keyword new** instead of **override**:

```

public class AfricanLion : Lion
{
    // ...

    public new string ToString()
    {
        return string.Format(

```

```
        "(AfricanLion, male: {0}, weight: {1})",
        this.Male, this.Weight);
    }

    // ...
}

public class OverrideExample
{
    static void Main()
    {
        AfricanLion africanLion = new AfricanLion(true, 80);
        string asAfricanLion = africanLion.ToString();
        string asObject = ((object)africanLion).ToString();
        Console.WriteLine(asAfricanLion);
        Console.WriteLine(asObject);
    }
}
```

This is the result:

```
(AfricanLion, male: True, weight: 80)
Chapter_20_OOP.AfricanLion
```

We notice that the implementation of **Object.ToString()** is invoked when we upcast **AfricanLion** to **object**. In other words, when we use the keyword **new**, we create a new method, which hides the old one. The old method can then only be called with an upcast.

What would happen, if we reverted to using the keyword **override** in the previous example? Take a look for yourself:

```
(AfricanLion, male: True, weight: 80)
(AfricanLion, male: True, weight: 80)
```

Surprising, isn't it? It turns out that when we override a method, we cannot access the old implementation even if we use upcasting. This is because there are no longer two **ToString()** methods, but rather only the one we overrode.

A method, which can be overridden, is called **virtual**. In .NET, methods are not **virtual** by default. If we want a method to be overridable, we can do so by including the keyword **virtual** in the declaration of the method.

The explicit instructions to the compiler that we want to override a method (by using **override**), is a protection against mistakes. If there's a typo in the method's name or the types of its parameters, the compiler will inform us

immediately of this mistake. It will know something is not right when it cannot find a method with the same signature in any of the base classes.

[Virtual Methods](#) are explained in details in the [section about polymorphism](#).

Transitive Properties of Inheritance

In mathematics, **transitivity** indicates transferability of relationships. Let's take the indicator "larger than" (>) as an example. If $A > B$ and $B > C$, we can **conclude** that $A > C$. This means that the relation "larger than" (>) is transitive, because we can unequivocally determine whether A is larger or smaller than C and vice versa.

If the class **Lion** inherits the class **Felidae** and the class **AfricanLion** inherits **Lion**, then this implies that **AfricanLion** inherits **Felidae**. Therefore, **AfricanLion** also has the property **Male**, which is defined in **Felidae**. This useful property allows a particular functionality to be defined in the most appropriate class.

Transitiveness – Example

Here is an example, which **demonstrates the transitive property of inheritance**:

TransitivenessExample.cs

```
public class TransitivityExample
{
    static void Main()
    {
        AfricanLion africanLion = new AfricanLion(true, 15);
        // Property defined in Felidae
        bool male = africanLion.Male;
        africanLion.Male = true;
    }
}
```

It is because of the transitive property of inheritance that we can be sure that all classes include the method **ToString()** and all other methods of **Object** regardless of which class they inherit.

Inheritance Hierarchy

If we try to describe all big cats, then, sooner or later, we will end up with a relatively large group of classes, which inherit one another. All these classes, combined with the base classes, form a **hierarchy** of big cat classes. The easiest way to describe such hierarchies is by using **class diagrams**. Let's take a look at what a "class-diagram" is.

Class Diagrams

A **Class Diagram** is one of several types of diagrams defined in UML. **UML (Unified Modeling Language)** is a notation for visualizing different processes and objects related to software development. We will talk about this further in the section on [UML notation](#). Now let's discuss **class diagrams**, because they are used to describe visually class hierarchies, inheritance and the structure of the classes themselves.

What is UML Class Diagram?

It is commonly accepted to draw class diagrams as **rectangles** with **name**, **attributes** (member variables) and **operations** (methods). The connections between them are denoted with various types of arrows.

Briefly, we will explain two pieces of UML terminology, so we can understand the examples more easily. The first one is **generalization**. Generalization is a term signifying the **inheritance of a class** or the **implementation of an interface** (we will explain [interfaces](#) shortly).

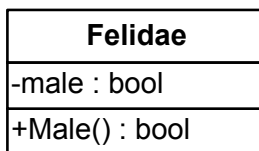
The other term is **association**. An association, would be, e.g. "The Lion has paws", where **Paw** is another class. Association is **has-a relationship**.



Generalization and association are the two main ways to reuse code.

A Class Based on a Class Diagram – Example

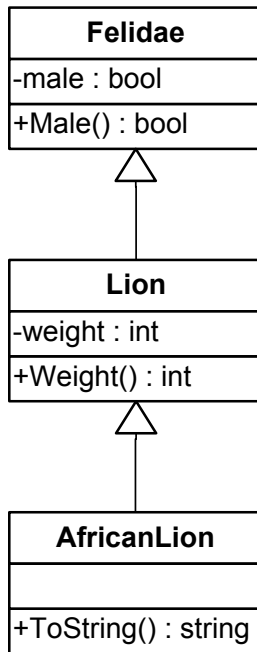
This is what a sample class diagram looks like:



The class is represented as a **rectangle**, divided in 3 boxes one under another. The **name** of the class is at the top. Next, there are the **attributes** (UML term) of the class (in .NET they are called member variables and properties). At the very bottom are the **operations** (UML term) or methods (in .NET jargon). The plus/minus signs indicate whether an attribute / operation is visible (+ means **public**) or not visible (- means **private**). **Protected** members are marked with #.

Class Diagram – Example of Generalization

Here is a class diagram that visually illustrates generalization (**Felidae** inherited by **Lion** inherited by **AfricanLion**):



In this example, the **arrows indicate generalization** (inheritance).

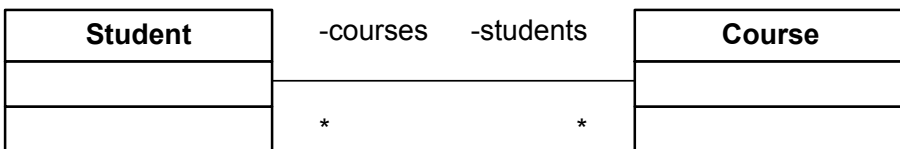
Associations

Associations denote **connections between classes**. They model mutual relations. They can define **multiplicity** (1 to 1, 1 to many, many to 1, 1 to 2, ..., and many to many).

A **many-to-many** association is depicted in the following way:

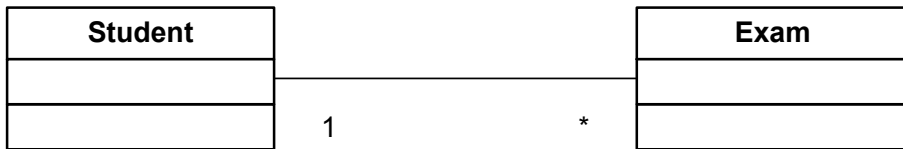


A **many-to-many association by attribute** is depicted in the following way:

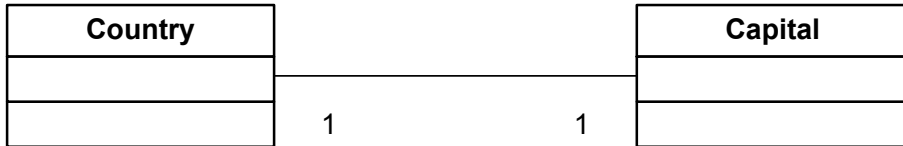


In this case, there are connecting attributes, which indicate the variables holding the connection between classes.

A **one-to-many** association is depicted like this:



A **one-to-one** association is depicted like this:



From Diagrams to Classes

Class diagrams are most often used for creating classes. Diagrams facilitate and speed up the **design of classes in a software project**.

We can create classes directly following the diagram above. Here is the **Capital** class:

```

Capital.cs

public class Capital { }
  
```

And the **Country** class:

```

Country.cs

public class Country
{
    /// <summary>Country's capital - association</summary>
    private Capital capital;

    // ...

    public Capital Capital
    {
        get { return capital; }
        set { this.capital = value; }
    }

    // ...
}
  
```

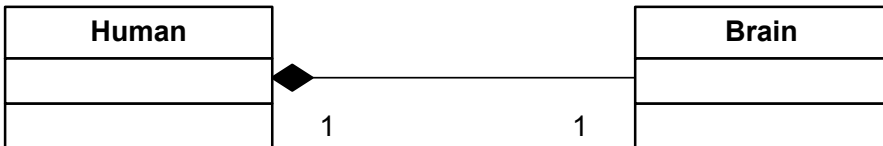
Aggregation

Aggregation is a special type of association. It models the **relationship of kind "whole / part"**. We refer to the parent class as an **aggregate**. The aggregated classes are called **components**. There is an empty rhombus at one end of the aggregation:



Composition

A filled rhombus represents composition. Composition is an aggregation where the **components cannot exist without the aggregate**:



Abstraction

The next core principle of object-oriented programming we are about to examine is "abstraction". **Abstraction means working with something we know how to use without knowing how it works internally**. A good example is a television set. We don't need to know the inner workings of a TV, in order to use it. All we need is a remote control with a small set of buttons (the interface of the remote) and we will be able to watch TV.

The same goes for objects in OOP. If we have an object **Laptop** and it needs a processor, we use the object **Processor**. We do not know (or rather it is of no concern to us) how it calculates. In order to use it, it's sufficient to call the method **Calculate()** with appropriate parameters.

Abstraction is something we do every day. This is an action, which obscures all details of a certain object that do not concern us and only uses the details, which are relevant to the problem we are solving. For example, in hardware configurations, there is an **abstraction called "data storage device"** which can be a **hard disk, USB memory stick or CD-ROM drive**. Each of these works in a different way internally but, from the point of view of the operating system and its applications, it is used in the same way – it **stores files and folders**. In Windows we have Windows Explorer and it can work with all devices in the same way, regardless of whether a device is a hard drive or a USB stick. It works with the **abstraction "storage device"** and is not involved with how data is read or written. The drivers of the particular device take care of that. They are implementations of the interface "data storage device".

Abstraction is one of the **most important concepts** in programming and OOP. It allows us to write **code, which works with abstract data structures** (like dictionaries, lists, arrays and others). We can work with an abstract data type by using its interface without concerning ourselves with its implementation. For instance, we can save to a file all elements from a list without bothering if it is implemented with an array, a linked list, etc. The code remains unchanged, when we work with other data types. We can even write new data types (we will discuss this later) and make them work with our program without changing it.

Abstraction allows us to do something very important – **define an interface for our applications**, i.e. to define **all tasks the program is capable to execute** and their respective input and output data. That way we can make a couple of small programs, each handling a smaller task. When we combine this with the ability to work with **abstract data**, we achieve great flexibility in integrating these small programs and much more opportunities for code reuse. These small subprograms are referred to as **components**. This approach for writing programs is widely adopted since it allows us to reuse not only objects, but entire subprograms as well.

Abstraction – Abstract Data Example

Here is an example, where we define a specific data type "African lion", but use it later on in an abstract manner through the "Felidae" abstraction. This **abstraction** does not concern itself with the details of all types of lions.

AbstractionExample.cs

```
public class AbstractionExample
{
    static void Main()
    {
        Lion lion = new Lion(true, 150);
        Felidae bigCat1 = lion;

        AfricanLion africanLion = new AfricanLion(true, 80);
        Felidae bigCat2 = africanLion;
    }
}
```

Interfaces

In the C# language the **interface** is a definition of a **role** (a group of abstract actions). It defines what sort of behavior a certain object must exhibit, without specifying how this behavior should be implemented. Interfaces are also known as **contracts** or **specifications** of behavior.

An object can have **multiple roles** (or **implement multiple interfaces** / contracts) and its users can utilize it from different points of view.

For example, an object of type **Person** can have the roles of **Soldier** (with behavior "shoot your enemy"), **Husband** (with behavior "love your wife") and **Taxpayer** (with behavior "pay your taxes"). However, every person implements its behavior in a different way; **John** pays his taxes on time, **George** pays them overdue and **Peter** doesn't pay them at all.

Some may ask why the base class of all objects (the class **Object**) is not an interface. The reason is because in such case, every class would have to implement a small, but very important group of methods and this would take an unnecessary amount of time. It turns out that not all classes need a specific implementation of **Object.GetHashCode()**, **Object.Equals(...)** and **Object.ToString()**, i.e. the default implementation suffices in most cases. It's not necessary to override any of the methods in the **Object** class, but if the situation calls for it we can. **Overriding methods** is explained in the [virtual methods section](#).

Interfaces – Key Concepts

An **interface can only declare methods and constants**.

A **method signature** is the combination of a method's **name** and a description of its **parameters** (type and order). In a class / interface all methods have to have different signatures and should not be identical with signatures of inherited methods.

A **method declaration** is the combination of a method's **return type and its signature**. The return type only specifies what the method returns.



A method is identified by its signature. The return type is not a part of it. If two methods' only difference is the return type (as in the case when a class inherits another), then it cannot be unequivocally decided which method must be executed.

A **class / method implementation** is the source code of a class / method. Usually it is between curly brackets: "{" and "}". Regarding methods, this is also referred to as the **method body**.

Interfaces – Example

An interface in .NET is defined with the **keyword interface**. An interface can contain only method declarations and constants. Here is an example of an interface:

Reproducible.cs

```
public interface Reproducible<T> where T : Felidae
{
```

```
T[] Reproduce(T mate);
}
```

We explained the generics in the "[Defining Classes](#)" chapter (section "[Generics](#)"). The interface we wrote has a method of type `T` (`T` must inherit `Felidae`) which returns an array of `T`.

And this is how the class `Lion`, which **implements the interface `Reproducible`** looks like:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
    // ...

    Lion[] Reproducible<Lion>.Reproduce(Lion mate)
    {
        return new Lion[]{new Lion(true, 12), new Lion(false, 10)};
    }
}
```

The name of the interface is coded in the declaration of the class (on the first row) and specifies the generic class.

We can indicate which method from a specific interface we implement by typing its name explicitly:

```
Lion[] Reproducible<Lion>.Reproduce(Lion mate)
```

In an interface, methods are only declared; the implementation is coded in the class implementing the interface, i.e. – `Lion`.

The class that implements a certain interface must **implement all methods in it**. The only exception is when the class is **abstract**. Then it can implement none, some or all of the methods. All remaining methods have to be implemented in some of the inheriting classes.

Abstraction and Interfaces

The best way to achieve abstraction is by **working through interfaces**. A component works with interfaces which another implements. That way, a change in the second component will not affect the first one as long as the new component implements the old interface. The interface is also called a **contract**. Every component upholds a certain contract (the signature of certain methods). That way, two components upholding a contract can communicate with each other without knowing how their counterpart works.

Some important interfaces from the Common Type System (CTS) are the **list** and **collection** interfaces: **System.Collections.Generic.IList<T>** and **System.Collections.Generic.ICollection<T>**. All of the standard .NET collection classes implement these interfaces and the various components pass different implementations (arrays, linked lists, hash tables, etc.) to one another using a common interface.

Collections are an excellent example of an object-oriented library with classes and interfaces that actively use all core principles of OOP: abstraction, inheritance, encapsulation and polymorphism.

When Should We Use Abstraction and Interfaces?

The answer to this question is: **always when we want to achieve abstraction of data or actions**, whose implementation can change later on. Code, which communicates with another piece of code through interfaces, is much more resilient to changes than code written using specific classes. **Working through interfaces is common and a highly recommended practice** – one of the basic rules for writing high-quality code.

When Should We Write Interfaces?

It is always a good idea to use interfaces **when functionality is exposed to another component**. In the interface we include only the functionality (in the form of a declaration) that others need to see.

Internally, a program / component can use interfaces for **defining roles**. That way, an object can be used by different classes through different roles.

Encapsulation

Encapsulation is one of the main concepts in OOP. It is also called "**information hiding**". An object has to provide its users only with the essential information for manipulation, without the internal details. A **Secretary** using a **Laptop** only knows about its screen, keyboard and mouse. Everything else is hidden internally under the cover. She **does not know about the inner workings of Laptop**, because she doesn't need to, and if she does, she might make a mess. Therefore parts of the properties and methods remain hidden to her.

The person writing the class has to decide what should be **hidden** and what not. When we program, we must define as **private** every method or field which other classes should not be able to access.

Encapsulation – Examples

The example below shows how to **hide methods** that the class' user doesn't have to be familiar with and are only used internally by the author of the class. First, we define an **abstract class Felidae**, which defines the public operations of cats (regardless of the cat's type):

Felidae.cs

```
public class Felidae
{
    public virtual void Walk()
    {
        // ...
    }

    // ...
}
```

This is how the class **Lion** looks like:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
    // ...

    private Paw frontLeft;
    private Paw frontRight;
    private Paw bottomLeft;
    private Paw bottomRight;

    private void MovePaw(Paw paw) {
        // ...
    }

    public override void Walk()
    {
        this.MovePaw(frontLeft);
        this.MovePaw(frontRight);
        this.MovePaw(bottomLeft);
        this.MovePaw(bottomRight);
    }

    // ...
}
```

The public method **Walk()** calls some other **private** method 4 times. That way the base class is short – it consists of a single method. The implementation, however, calls another of its methods, which is hidden from the users of the class. That way, **Lion doesn't publicly disclose information about its inner workings** (it encapsulates certain behavior).

At a later stage, this makes it possible to change its implementation without any of the other classes finding out and requiring changes.

Polymorphism

The next fundamental principle of Object-Oriented Programming is "Polymorphism". **Polymorphism allows treating objects of a derived class as objects of its base class.** For example, big cats (base class) catch their prey (a method) in different ways. A Lion (derived class) sneaks on it, while a Cheetah (another derived class) simply outruns it.

Polymorphism allows us to treat a cat of random size just like a big cat and command it "catch your prey", regardless of its exact size.

Polymorphism can bear strong resemblance to abstraction, but it is mostly related to **overriding methods in derived classes**, in order **to change their original behavior** inherited from the base class. Abstraction is associated with creating an interface of a component or functionality (defining a role). We are going to explain method overriding shortly.

Abstract Classes

What happens if we want to specify that the class **Felidae** is **incomplete** and only its successors can have instances? This is accomplished by putting **the keyword abstract** before the name of the class and indicates that the class is **not ready to be instantiated**. We refer to such classes as **abstract classes**. And how do we indicate which exact part of the class is incomplete? Once again, this is accomplished by putting the keyword **abstract** before the name of the method to be implemented. This method is called an **abstract method** and cannot have an implementation, but a declaration only.

Each class with **at least one abstract method** must be abstract. Makes sense, right? However, the opposite is not true. It is possible to define a class as an abstract one, even when there are no abstract methods in it.

Abstract classes are something **in the middle between classes and interfaces**. They can define **ordinary methods** and **abstract methods**. Ordinary methods have an implementation, whereas abstract methods are **empty** (without an implementation) and remain to be implemented later by the derived classes.

Abstract Class – Examples

Let's take a look at an **example of an abstract class**:

Felidae.cs

```
/// <summary>Latin for "cats"</summary>
public abstract class Felidae
{
```



```
// ...  
  
protected void Hide()  
{  
    // ...  
}  
  
protected void Run()  
{  
    // ...  
}  
  
public abstract bool CatchPrey(object prey);  
}
```

Notice how in the example above the ordinary methods **Hide()** and **Run()** have a body, while the abstract method **CatchPrey()** does not. Notice that the methods are declared as **protected**.

Here is how the **implementation** of the above abstraction looks like:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>  
{  
    protected void Ambush()  
    {  
        // ...  
    }  
  
    public override bool CatchPrey(object prey)  
    {  
        base.Hide();  
        this.Ambush();  
        base.Run();  
        // ...  
        return false;  
    }  
}
```

Here is one more example of **abstract behavior**, implemented with an abstract class and a polymorphic call to an abstract method. In this example we define abstract method and we override it later in a descendant class. Let's see the code and discuss it later.

Firstly, we define the abstract class **Animal**:

Animal.cs

```
public abstract class Animal
{
    public void PrintInformation()
    {
        Console.WriteLine("I am a {0}.", this.GetType().Name);
        Console.WriteLine(GetTypicalSound());
    }

    protected abstract String GetTypicalSound();
}
```

We also define the class **Cat**, which **inherits the abstract class Animal** and defines an implementation of the abstract method **GetTypicalSound()**:

Cat.cs

```
public class Cat : Animal
{
    protected override String GetTypicalSound()
    {
        return "Meooooow!";
    }
}
```

If we execute the following program:

```
public class AbstractClassExample
{
    static void Main()
    {
        Animal cat = new Cat();
        cat.PrintInformation();
    }
}
```

we are going to get the following result:

```
I am a Cat.
Meooooow!
```

In the example, the **PrintInformation()** method from the abstract class does its work by relying on the result from a call to the **abstract method GetTypicalSound()** which is expected to be implemented in different ways by

the kinds of animals (the various successors of the class **Animal**). Different animals make distinct sounds, but the functionality for printing information about animals is common to all animals, and that's why it is exported to the base class.

Purely Abstract Classes

Abstract classes, as well as interfaces, **cannot be instantiated**. If we try to create an instance of an abstract class, we are going to get an error during compilation.



Sometimes a class can be declared abstract, even if it has no abstract methods, in order to simply prohibit using it directly without creating an instance of a successor.

A **pure abstract class** is an abstract class, which has no implemented methods and no member variables. It is **very similar to an interface**. The fundamental difference is that a class can implement many interfaces and inherit only one class (even if that class is abstract).

Initially, interfaces were not necessary in the presence of "multiple inheritance". They had to be conceived as a means to supersede it in specifying the numerous roles of an object.

Virtual Methods

A method, which can be overridden in a derived class, is called a **virtual method**. Methods in .NET by default aren't virtual. If we want to make a method virtual, we mark it with **the keyword virtual**. Then the derived class can declare and define a method with the same signature.

Virtual methods are important for **method overriding**, which lies at the heart of polymorphism.

Virtual Methods – Example

We have a class inheriting another and the two classes share a common method. Both versions of the method write on the console. Here is how the **Lion** class looks like:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
    public override void CatchPrey(object prey)
    {
        Console.WriteLine("Lion.CatchPrey");
    }
}
```

Here is how the **AfricanLion** class looks like:

```


AfricanLion.cs


public class AfricanLion : Lion
{
    public override void CatchPrey(object prey)
    {
        Console.WriteLine("AfricanLion.CatchPrey");
    }
}

```

We make three attempts to create instances and call the method **CatchPrey**.

```


VirtualMethodsExample.cs


public class VirtualMethodsExample
{
    static void Main()
    {
        Lion lion = new Lion(true, 80);
        lion.CatchPrey(null);
        // Will print "Lion.CatchPrey"

        AfricanLion lion = new AfricanLion(true, 120);
        lion.CatchPrey(null);
        // Will print "AfricanLion.CatchPrey"

        Lion lion = new AfricanLion(false, 60);
        lion.CatchPrey(null);
        // Will print "AfricanLion.CatchPrey", because
        // the variable lion has a value of type AfricanLion
    }
}

```

In the last attempt, you can clearly see how, in fact, **the overwritten method is called** and not the base method. This happens, because it is validated what the actual class behind the variable is and whether it implements (overwrites) that method. **Rewriting of methods** is also called **overriding of virtual methods**.

Virtual methods as well as abstract methods can be overridden. Abstract methods are actually virtual methods without a specific implementation. All methods defined in an interface are abstract and therefore virtual, although this is not explicitly defined.

Virtual Methods and Methods Hiding

In the example above, the implementation of the base class is hidden and omitted. Here is how we can also use it as part of the new implementation (in case we want to complement the old implementation rather than override it).

Here is how the **AfricanLion** class looks like:

AfricanLion.cs
<pre>public class AfricanLion : Lion { public override void CatchPrey(object prey) { Console.WriteLine("AfricanLion.CatchPrey"); Console.WriteLine("calling base.CatchPrey"); Console.Write("\t"); base.CatchPrey(pre); Console.WriteLine("...end of call."); } }</pre>

In this example, three lines will be written on the console when **AfricanLion.CatchPrey(...)** is called:

<pre>AfricanLion.CatchPrey calling base.CatchPrey Lion.CatchPrey ...end of call.</pre>
--

The Difference between Virtual and Non-Virtual Methods

Some may ask what the difference between the virtual and non-virtual methods is.

Virtual methods are used when we expect from derived classes to change / complement / **alter some of the inherited functionality**. For example, the method **Object.ToString()** allows derived classes to change / replace its implementation in any way they want. Then, even if we work with an object not directly, but rather by upcasting it to **Object**, we use the overwritten implementation of the virtual methods.

Virtual methods are a key characteristic of objects when we talk about [abstraction](#) and working with abstract types.

Sealing of methods is done when we rely on a piece of functionality and we don't want it to be altered. We already know that methods are **sealed** by default. But if we want a base class' virtual method to become sealed in a derived class, we use **override sealed**.

The **string class has no virtual methods**. In fact, inheriting **string** is entirely **forbidden for inheritance** through the keyword **sealed** in its declaration. Here are parts of the declarations of **string** and **object** classes (the ellipses in square brackets indicate omitted, irrelevant code):

```
namespace System
{
    [...] public class Object
    {
        [...] public Object();
        [...] public virtual bool Equals(object obj);
        [...] public static bool Equals(object objA, object objB);
        [...] public virtual int GetHashCode();
        [...] public Type GetType();
        [...] protected object MemberwiseClone();
        [...] public virtual string ToString();
    }

    [...] public sealed class String : [...]
    {
        [...] public String(char* value);
        [...] public int IndexOf(string value);
        [...] public string Normalize();
        [...] public string[] Split(params char[] separator);
        [...] public string Substring(int startIndex);
        [...] public string ToLower(CultureInfo culture);
        [...]
    }
}
```

When Should We Use Polymorphism?

The answer to this question is simple: whenever we want **to enable changing a method's implementation in a derived class**. It's a good rule to work with the most basic class possible or directly with an interface. That way, changes in used classes reflect to a much lesser extent on classes written by us. The less a program knows about its surrounding classes, the fewer changes (if any) it would have to undergo.

Cohesion and Coupling

The terms **cohesion and coupling** are inseparable from OOP. They complement and explain further some of the principles we have described so far. Let's get familiar with them.

Cohesion

The concept of **cohesion** shows to what degree a program's or a component's various tasks and responsibilities are **related to one another**, i.e. how much a program is focused on solving a single problem. Cohesion is divided into **strong cohesion** and **weak cohesion**.

Strong Cohesion

Strong cohesion indicates that the responsibilities and tasks of a piece of code (a method, class, component or a program) are **related to one another** and intended to **solve a common problem**. This is something we must always aim for. Strong cohesion is a typical characteristic of high-quality software.

Strong Cohesion in a Class

Strong cohesion in a class indicates that the class **defines only one entity**. We mentioned earlier that an entity can have many roles (Peter is a soldier, husband and a taxpayer). Each of these roles is defined in the same class. Strong cohesion indicates that the class solves only one task, one problem, and not many at the same time.

A class, which does **many things at the same time**, is difficult to understand and maintain. Consider a class, which implements a hash table, provides functions for printing, sending an e-mail and working with trigonometric functions all at once. How do we name such a class? If we find it difficult to answer this question, this means that we have failed to achieve **strong cohesion** and have to separate the class into several smaller classes, each solving a single task.

Strong Cohesion in a Class – Example

As an example of strong cohesion we can point out the **System.Math** class. It performs a **single task**: it provides mathematical calculations and constants:

- **Sin(), Cos(), Asin()**
- **Sqrt(), Pow(), Exp()**
- **Math.PI, Math.E**

Strong Cohesion in a Method

A method is well written when it performs only one task and performs it well. A method, which does a lot of work related to different things, has **bad cohesion**. It has to be **broken down into simpler methods**, each solving only one task. Once again, the question is posed what name should we give to a method, which finds prime numbers, draws 3D graphics on the screen, communicates with the network and prints records extracted from a data base? Such a method has **bad cohesion** and has to be logically separated into several methods.

Weak Cohesion

Weak cohesion is observed along with **methods, which perform several unrelated tasks**. Such methods take several different groups of parameters, in order to perform different tasks. Sometimes, this requires logically unrelated data to be unified for the sake of such methods. Weak cohesion is harmful and must be avoided!

Weak Cohesion – Example

Here is a sample class with weak cohesion:

```
public class Magic
{
    public void PrintDocument(Document d) { ... }
    public void SendEmail(string recipient,
        string subject, string text) { ... }
    public void CalculateDistanceBetweenPoints(
        int x1, int y1, int x2, int y2) { ... }
}
```

Best Practices with Cohesion

Strong cohesion is quite logically the "good" way of writing code. The concept is associated with simpler and clearer source code – code that is easier to maintain and reuse (because of the fewer tasks it has to perform).

Contrarily, with **weak cohesion** each change is a ticking time bomb, because it could affect other functionality. Sometimes a logical task is spread out to several different modules and thus changing it is more labor intensive. Code reuse is also difficult, because a component does several unrelated tasks and to reuse it the exact same conditions must be met which is hard to achieve.

Coupling

Coupling mostly describes the extent to which components / classes **depend on one another**. It is broken down into **loose coupling** and **tight coupling**. Loose coupling usually correlates with strong cohesion and vice versa.

Loose Coupling

Loose coupling is defined by a piece of code's (program / class / component) communication with other code through **clearly defined interfaces** (contracts). A change in the implementation of a loosely coupled component doesn't reflect on the others it communicates with. When you write source code, you **must not rely on inner characteristics** of components (specific behavior that is not described by interfaces).

The contract has to be maximally simplified and define only the required behavior for this component's work by hiding all unnecessary details.

Loose coupling is a code characteristic you should aim for. It is one of the characteristics of high-quality programming code.

Loose Coupling – Example

Here is an **example of loose coupling** between classes and methods:

```
class Report
{
    public bool LoadFromFile(string fileName) { ... }
    public bool SaveToFile(string fileName) { ... }
}

class Printer
{
    public static int Print(Report report) { ... }
}

class Example
{
    static void Main()
    {
        Report myReport = new Report();
        myReport.LoadFromFile("DailyReport.xml");
        Printer.Print(myReport);
    }
}
```

In this example, **none of the methods depend on the others**. The methods rely only on some of the parameters, which are passed to them. Should we need one of the methods in a next project, we could easily take it out and reuse it.

Tight Coupling

We achieve tight coupling when there are many input parameters and output parameters; when we use undocumented (in the contract) characteristics of another component (for example, a dependency on static fields in another class); and when we use many of the so called control parameters that indicate behavior with actual data. **Tight coupling** between two or more methods, classes or components means that **they cannot work independently of one another** and that a change in one of them will also affect the rest. This leads to difficult to read code and big problems with its maintenance.

Tight Coupling – Example

Here is an **example of tight coupling** between classes and methods:

```
class MathParams
{
    public static double operand;
    public static double result;
}

class MathUtil
{
    public static void Sqrt()
    {
        MathParams.result = CalcSqrt(MathParams.operand);
    }
}

class SpaceShuttle
{
    static void Main()
    {
        MathParams.operand = 64;
        MathUtil.Sqrt();
        Console.WriteLine(MathParams.result);
    }
}
```

Such code is **difficult to understand and maintain**, and the likelihood of mistakes when using it is great. Think about what happens if another method, which calls `Sqrt()`, passes its parameters through the same static variables `operand` and `result`.

If we have to use the same functionality for deriving square root in a subsequent project, we will not be able to simply copy the method `Sqrt()`, but rather we will have to copy the classes `MathParams` and `MathUtil` together with all of their methods. This makes the code difficult to reuse.

In fact, the above code is an **example of bad code** according to all rules of Procedural and Object-Oriented Programming and if you think twice, you will certainly identify at least several more disregarded recommendations from those we have given you so far.

Best Practices with Coupling

The most common and advisable way of invoking a well written module's functionality is through interfaces. That way, the functionality can be substituted without clients of the code requiring changes. The jargon expression for this is "**programming against interfaces**".

Most commonly, an interface describes a "**contract**" observed by this module. It is good practice not to rely on anything else other than what's described by

this contract. The use of inner classes, which are not part of the public interface of a module, is not recommended because their implementation can be substituted without substituting the contract (we already discussed this in the section "[Abstraction](#)").

It is good practice that the **methods are made flexible** and ready to work with all components, which observe their interfaces, and not only with definitive ones (i.e. to have implicit requirements). The latter would mean that these methods expect something specific from the components they can work with. It is also good practice that **all dependencies are clearly described and visible**. Otherwise, the maintenance of such code becomes difficult (it is riddled with stumbling-blocks).

A good example of strong cohesion and loose coupling we can find in the classes from the standard namespaces **System.Collections** and **System.Collections.Generic**. These .NET classes for working with collections have **strong cohesion**. Each solves a single problem and allows easy reuse. These classes have another characteristic of high-quality programming code: **loose coupling**. The classes, implementing the collections, are not related to one another. Each works **through a strictly defined interface** and **does not give away details of its implementation**. All methods and fields not from the interface are hidden, in order to reduce the possibility of coupling with them. Methods in the collection classes do not depend on static variables and do not rely on any input data except for their inner state and passed parameters. This is good practice every programmer sooner or later attains with gained experience.



Spaghetti Code

Spaghetti code is **unstructured code with unclear logic**; it is **difficult to read, understand and maintain**; it violates and mixes up consistency; it has **weak cohesion and tight coupling**. Such code is associated with spaghetti, because it is just as tangled and twisted. When you pull out a strand of spaghetti (i.e. a class or method), the whole dish of spaghetti can turn out tangled in it (i.e. changes in one method or class lead to dozens of other changes because of the strong dependence between them). It is almost impossible to reuse spaghetti code, since there is no way to separate that part of the code, which is practically applicable.

Spaghetti code is achieved when you have written code, supplement it and have to readapt it again and again every time the requirements change. Time passes by until a moment comes when it has to be rewritten from scratch.

Cohesion and Coupling in Engineering Disciplines

If you think that the principles of strong cohesion and loose coupling apply only to programming, you are deeply mistaken. These are **fundamental engineering principles** you will come across in construction, machine building, electronics and thousands of other fields.

Let's take, for instance, a **hard disk drive (HDD)**:



It **solves only one task** doesn't it? The hard disk solves the task of storing data. It does not cool down the computer, does not make sounds, has no computing power and is not used as a keyboard. It is connected to the computer with two cables only, i.e. it has a **simple interface** for access and is not bound to other peripherals. The hard disk **works separately** and other devices aren't concerned about how it works exactly. The CPU commands it to "read" and it reads, then it commands it to "write" and it writes. How exactly

it does this remains **hidden inside** it. Different models can work in different ways, but that is their own concern. You can see that the CPU has strong cohesion, loose coupling, good abstraction and good encapsulation. This is how you should implement your classes – they must **do only one thing, do it well, bind them minimally to other classes** (or not link them at all whenever that's possible), have a clear interface and good abstraction and to hide the details of their internal workings.

Here is another example: imagine what would happen, if the processor, the hard disk, the CD-ROM drive and the keyboard were soldered to the motherboard of the computer. It would mean that if any part of the keyboard were broken, you would have to throw away the whole computer. You can see how hardware cannot work well with tight coupling and weak cohesion. The same applies to software.

Object-Oriented Modeling (OOM)

Suppose we have a problem or task to solve. The problem usually comes from the real world. It exists in a reality we are going to call its surrounding environment.

Object-oriented modeling (OOM) is a process associated with OOP where all objects related to the problem we are solving are brought out (a model is created). Only the classes' characteristics, which are important for solving this particular problem, are elicited. The rest are ignored. That way, we create a new reality, a **simplified version of the original** one (its model), such that it allows us to solve the problem or task.

For example, if we model a **ticketing system**, the **important characteristics** of a passenger could be their name, their age, whether they use a discount and whether they are male or female (if we sell sleeping

berths). A passenger has **many other not important characteristics we aren't concerned about**, such as the color of their eyes, what shoe size they wear, what books they like or what beer they drink.

By modeling, **a simplified model of reality is created** in order to solve a specific task. In object-oriented modeling, the model is created by means of OOP: via classes, class attributes, class methods, objects, relations between classes, etc. Let's scrutinize this process.

Steps in Object-Oriented Modeling

Object-oriented modeling is usually performed in these steps:

- Identification of classes.
- Identification of class attributes.
- Identification of operations on classes.
- Identification of relations between classes.

We will consider a short **example** through which we will demonstrate how to apply these steps.

Identification of Classes

Suppose we have the following excerpt from a system's **specification**:

The user must be able to describe each product by its characteristics, including name and product number. If the barcode doesn't match the product, an error must be generated on the error screen. There has to be a daily report for all transactions specified in section 9.3.

Here is how we identify key concepts:

The **user** must be able to describe each **product** by its **characteristics**, including **name** and **product number**. If the **barcode** doesn't match the product, an **error** must be generated on the **error screen**. There has to be a **daily report** for all **transactions** specified in section 9.3.

We have just **identified the classes** we will need. The names of the classes are **the nouns in the text**, usually common nouns in singular like **Student**, **Message**, **Lion**. Avoid names that don't come from the text, such as: **StrangeClass**, **AddressTheStudentHas**.

Sometimes it's difficult to determine whether some subject or phenomena from the real world has to be a class. For example, the **address** can be defined as a **class Address or a string**. The better we explore the problem, the easier it will be to decide which entities must be represented as classes. When a class becomes large and complicated it has to be broken down into several smaller classes.

Identification of Class Attributes

Classes have **attributes (characteristics)**, for example the class **Student** has a name, institution and a list of courses. Not all characteristics are important for a software system. For example, as far as the class **Student** is concerned eye color is a non-essential characteristic. **Only essential characteristics have to be modeled.**

Identification of Operations on Classes

Each class must have **clearly defined responsibilities** – what objects or processes from the real world it identifies and what tasks it performs. Each action in the program is performed by one or several methods in some class. The actions are modeled as operations (methods).

A combination of **verb + noun is used for the name of a method**, e.g. **PrintReport()**, **ConnectToDatabase()**. We cannot define all methods of a given class immediately. Firstly, we define the most important methods – those that implement the basic responsibilities of the class. Over time additional methods appear.

Identification of Relationships between Classes

If a student is from a faculty and this is important for the task we are solving, then student and faculty are related, i.e. the **Faculty** class has a list of **Students**. These relations are called **associations** (remember the "[Class Diagrams](#)" section).

UML Notation

UML (Unified Modeling Language) was mentioned in the [section about inheritance](#) where we discussed class diagrams. The UML notation defines several additional types of diagrams. Let's check out some of them briefly.

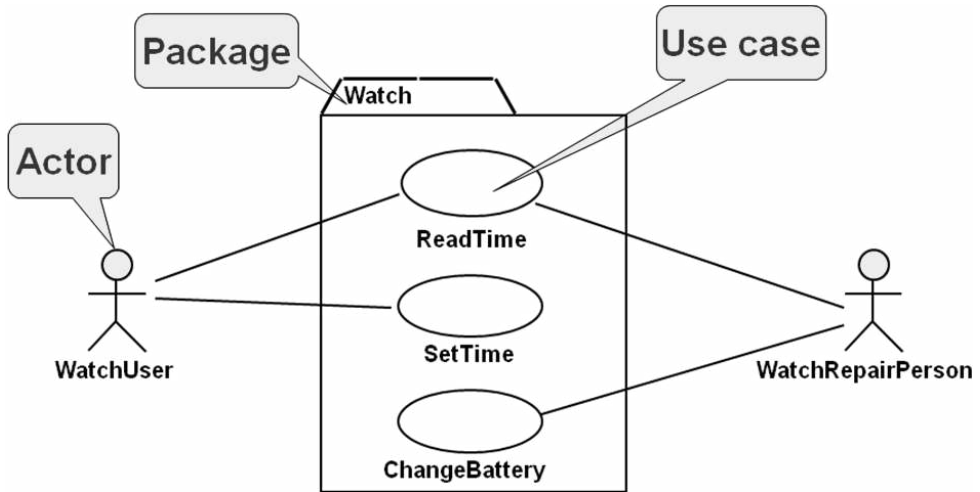
Use Case Diagrams

They are used when we elicit the requirements for the description of possible actions. **Actors** represent roles (types of users).

Use cases describe interaction between the actors and the system. The use case model is a group of use cases – it provides a complete description of a system's functionality.

Use Case Diagrams – Example

Here is how a **use case diagram** looks like:



The **actor** (the “dwarf” in the diagram) is someone who interacts with the system (a user, external system or, for instance, an external environment). The **actor has a unique name** and, possibly, a **description**. In our case actors are the **WatchUser** and the **WatchRepairPerson**.

A **use case** (the “egg” in the diagram) describes a **single functionality of the system**, a single **action** that can be performed by some actor. It has a **unique name** and is **related to actors**. It can have input and output conditions. Most frequently, it contains a flow of operations (a process). It can also have other requirements. We have three use cases in the diagram above: **ReadTime**, **SetTime** and **ChangeBattery**.

A **package** holds several logically related use cases.

Lines connect actors to the use cases they perform. An actor can perform or be involved in one or several use cases.

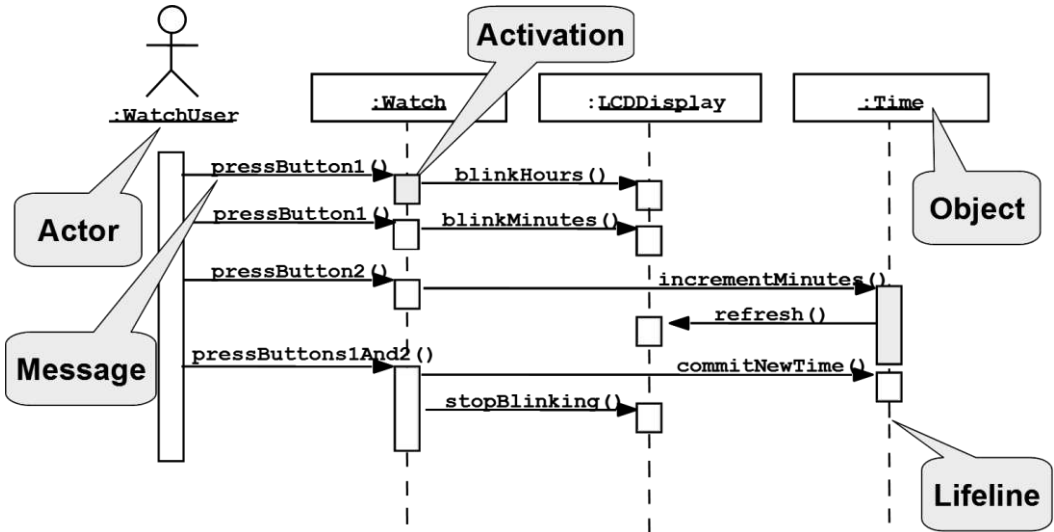
Sequence Diagrams

Sequence diagrams are used when modeling the requirements of **process specification** and describing use case scenarios more extensively. They allow describing additional participants in the processes and the sequence of the actions over the time. They are used in designing the descriptions of system interfaces.

Sequence diagrams describe **what happens over the time**, the interactions over the time, the **dynamic view** over the system, a **sequence of steps**, just like an algorithm.

Sequence Diagrams – Example

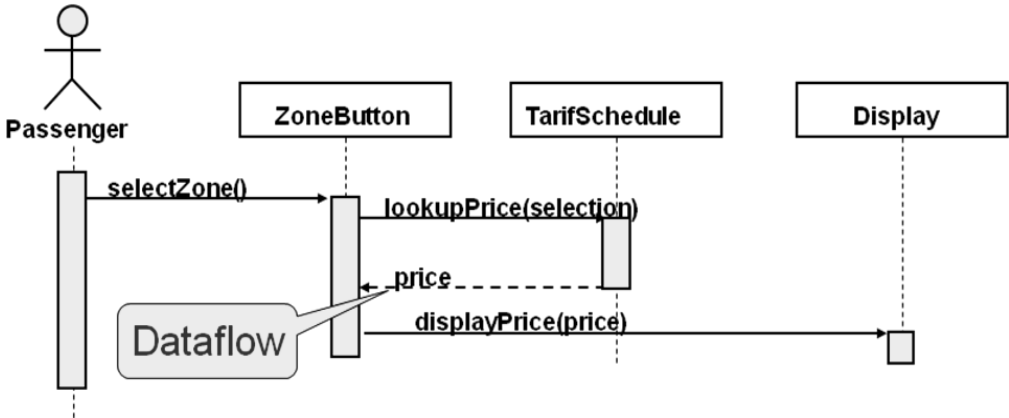
Here is how a sequence diagram looks like:



Classes are depicted with columns (lifelines). **Messages (actions)** are depicted with arrows and text above the arrows. **Participants** are depicted with wide rectangles. **States** are depicted with dashed lines. The period of activity (**activation**) of certain class during the time is depicted as narrow rectangles.

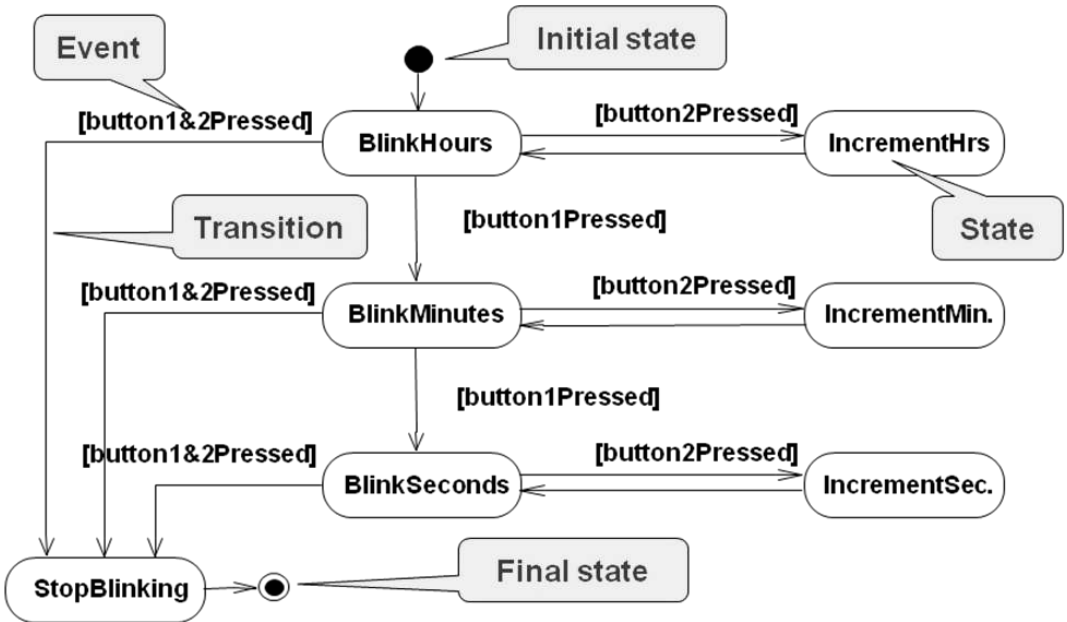
Messages – Example

The direction of the arrow designates the **sender** and the **recipient** of a **message** (a method call in OOP). Horizontal dashed lines depict **data flow**:



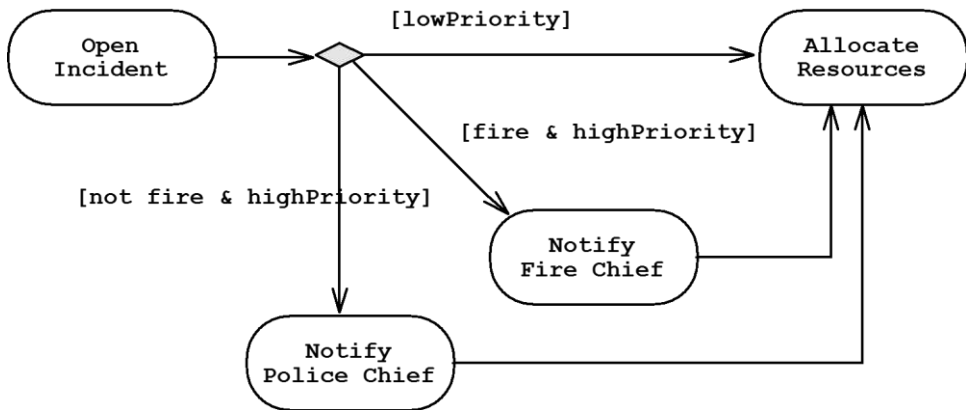
Statechart Diagrams

Statechart diagrams describe the possible **states** of certain process and the possible **transitions** between them along with the conditions for the transitions. They represent **finite-state automata (state machines)**. Below we have an **example of statechart diagram** that illustrates the states and transitions of typical process of changing the current time of a wall clock which has two buttons and a screen:



Activity Diagrams

Activity diagrams are a **special type of statechart diagrams** where **conditions are actions**. They show the flow of actions in a system:

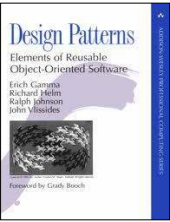


Design Patterns

Few years after the onset of the object-oriented paradigm it was found that there are many situations, which occur frequently during software development, such as a class, which must have only one instance within the entire application.

Design patterns appeared as proven and highly-efficient **solutions to the most common problems of object-oriented modeling**. Design patterns are systematically described in the eponymous book by Erich Gamma & Co.

"**Design Patterns: Elements of Reusable Object-Oriented Software**" (ISBN 0-201-63361-2). The patterns in this book are called "**the GoF patterns**" or "classical design patterns".



This is one of the few books in the field of computer science, which remain current 15 years after publishing. Design patterns complement the basic principles of OOP with **well-known solutions of well-known problems**. A good place to start studying the design patterns is their Wikipedia article: [en.wikipedia.org/wiki/Design_pattern_\(computer_science\)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)). You may also check the "Data & Object Factory" **patterns catalog** <http://www.dofactory.com/Patterns/Patterns.aspx>, where the authors provide C# implementation of the classical GoF patterns.

The Singleton Design Pattern

This is the most popular and most frequently used design pattern. It allows a **class to have only one instance** and defines where it has to be taken from. Typical examples are classes, which define references to singular entities (a virtual machine, operating system, window manager in a graphical application or a file system) as well as classes of the next pattern (factory).

The Singleton Design Pattern – Example

Here is a sample implementation of the **singleton** design pattern:

Singleton.cs

```
public class Singleton
{
    // The single instance
    private static Singleton instance;

    // Initialize the single instance
    static Singleton()
    {
        instance = new Singleton();
    }

    // The property for retrieving the single instance
    public static Singleton Instance
    {
        get { return instance; }
    }

    // Private constructor: protects against direct instantiation
    private Singleton() { }
}
```

We have a **hidden (private) constructor** in order to limit external instantiations. We have a static variable, which holds **the only instance**. We initialize it only once in the **static constructor** of the class. The property for retrieving the single instance is usually called **Instance**.

The pattern can undergo many optimizations, such as the so called "**lazy initialization**" of the only variable, in order to save memory, but this is its classical form.

The Factory Method Design Pattern

Factory method is another very common design pattern. It is intended for "**producing**" **objects**. The instantiation of an object is not performed directly, but rather by the factory method. This allows the factory method to decide which specific instance to create from a family of classes implementing a common interface. The solution can depend on the environment, a parameter or some system setting.

The Factory Method Design Pattern – Example

Factory methods **encapsulate object creation**. This is useful if the creation process is very complicated – if it depends on settings in configuration files or input data by the user.

Suppose we have a class which contains graphics files (**png, jpeg, bmp**, etc.) and creates reduced size copies of them (the so called **thumbnails**). A variety of formats are supported, each represented by a class:

```
public class Thumbnail
{
    // ...
}

public interface Image
{
    Thumbnail CreateThumbnail();
}

public class GifImage : Image
{
    public Thumbnail CreateThumbnail()
    {
        // ... Create a GIF thumbnail here ...
        return gifThumbnail;
    }
}

public class JpegImage : Image
```

```
{
    public Thumbnail CreateThumbnail()
    {
        // ... Create a JPEG thumbnail here ...
        return jpegThumbnail;
    }
}
```

Here is how the class holding an **album of images** looks like:

```
public class ImageCollection
{
    private IList<Image> images;

    public ImageCollection(IList<Image> images)
    {
        this.images = images;
    }

    public IList<Thumbnail> CreateThumbnails()
    {
        IList<Thumbnail> thumbnails =
            new List<Thumbnail>(images.Count);
        foreach (Image thumb in images)
        {
            thumbnails.Add(thumb.CreateThumbnail());
        }
        return thumbnails;
    }
}
```

The **client** of the program may require thumbnails of all images in the album:

```
public class Example
{
    static void Main()
    {
        IList<Image> images = new List<Image>();

        images.Add(new JpegImage());
        images.Add(new GifImage());

        ImageCollection imageRepository =
            new ImageCollection(images);
    }
}
```

```
        Console.WriteLine(imageRepository.CreateThumbnails());
    }
}
```

Other Design Patterns

There are dozens of other well-known design patterns, but we are not going to discuss them. The more inquisitive readers can look up "Design Patterns" on the internet and learn what other design patterns, such as **Abstract Factory**, **Prototype**, **Adapter**, **Composite**, **Facade**, **Command**, **Observer**, **Iterator**, etc. serve for and how they are put into use. If you pursue .NET development more seriously, you will see for yourselves that the whole standard library (FCL) is built on the principles of OOP and the classic design patterns are very actively used.

Exercises

1. We are given a **school**. The school has classes of students. Each class has a set of **teachers**. Each teacher teaches a set of **courses**. The students have a name and unique number in the class. **Classes** have a unique text identifier. Teachers have names. Courses have a name, count of classes and count of exercises. The teachers as well as the students are people. Your task is to model the classes (in terms of OOP) along with their attributes and operations define the class hierarchy and create a class diagram with Visual Studio.
2. Define a class **Human** with properties "first name" and "last name". Define the class **Student** inheriting **Human**, which has the property "mark". Define the class **Worker** inheriting **Human** with the property "wage" and "hours worked". Implement a "calculate hourly wage" method, which calculates a worker's hourly pay rate based on wage and hours worked. Write the corresponding constructors and encapsulate all data in properties.
3. Initialize an array of 10 students and sort them by mark in ascending order. Use the interface **System.IComparable<T>**.
4. Initialize an array of 10 workers and sort them by salary in descending order.
5. Define an abstract class **Shape** with abstract method **CalculateSurface()** and fields **width** and **height**. Define two additional classes for a **triangle** and a **rectangle**, which implement **CalculateSurface()**. This method has to return the areas of the rectangle (**height*width**) and the triangle (**height*width/2**). Define a class for a **circle** with an appropriate constructor, which initializes the two fields (**height** and **width**) with the same value (the radius) and implement the abstract method for calculating the area. Create an array of different shapes and calculate the area of each shape in another array.

6. Implement the following classes: **Dog**, **Frog**, **Cat**, **Kitten** and **Tomcat**. All of them are animals (**Animal**). Animals are characterized by **age**, **name** and **gender**. Each animal makes a sound (use a virtual method in the **Animal** class). Create an array of different animals and print on the console their name, age and the corresponding sound each one makes.
7. Using Visual Studio generate the **class diagrams** of the classes from the previous task with it.
8. A **bank** holds different **types of accounts** for its customers: **deposit** accounts, **loan** accounts and **mortgage** accounts. Customers can be **individuals** or **companies**. All accounts have a customer, balance and interest rate (monthly based). **Deposit accounts** allow depositing and withdrawing of money. **Loan and mortgage accounts** allow only depositing. All accounts can calculate their interest for a given period (in months). In the general case, it is calculated as follows: **number_of_months * interest_rate**. **Loan accounts** have no interest rate during the first 3 months if held by individuals and during the first 2 months if held by a company. **Deposit accounts** have no interest rate if their balance is positive and less than 1000. **Mortgage accounts** have $\frac{1}{2}$ the interest rate during the first 12 months for companies and no interest rate during the first 6 months for individuals. Your task is to write an object-oriented model of the bank system. You must identify the classes, interfaces, base classes and abstract actions and implement the interest calculation functionality.
9. Read about the **Abstract Factory** design pattern and implement it in C#.

Solutions and Guidelines

1. The task is trivial. Just follow the problem description and write the code.
2. The task is trivial. Just follow the problem description and write the code.
3. Implement **IComparable<T>** in **Student** and then sort the array.
4. This problem is like the previous one.
5. Just implement the classes as described in the problem description.
6. Printing information can be implemented in the virtual method **System.Object.ToString()**. In order to print the content of an array of animals, you can use a **foreach** loop.
7. If you have the full version of **Visual Studio**, just use "**Add New Item**" → "**Class Diagram**". Class diagrams are not supported in VS Express Edition. In this case you can find some other UML tool (see http://en.wikipedia.org/wiki/List_of_UML_tools).
8. Use abstract class **Account** with abstract method **CalculateInterest(...)**.
9. You can read about the "**abstract factory**" design pattern in Wikipedia: http://en.wikipedia.org/wiki/Abstract_factory_pattern.