Object-Oriented Programming with C#

Object-Oriented Programming, Part II

By Per Laursen
09-08-2022

INTRODUCTION **3**

The Has-a relationship (Composition) 3

The Is-a relationship (Inheritance) 3

Next-level parameterization 4

INHERITANCE **5**

The protected access level 5

Constructors in derived classes 6

Overriding methods 7

Polymorphic behavior 9

Calling base class methods 11

Abstract methods (and classes) 12

Interfaces 13

The Object class 15

EXCEPTIONS **16**

Throwing and catching 17

Rethrowing an exception 19

Exceptions summary 20

GENERICS – TYPES AS PARAMETERS **21**

Shortcomings of inheritance 21

Using types as parameters 23

Type constraints 25

Type parameter variance 27

The IComparable<T> and IComparer<T> interfaces 31

Generic methods 34

FUNCTIONS AS PARAMETERS **36**

A first attempt at function parameterization 36

Lambda expressions 39

Delegates 42

Events 44

EXERCISES 47

OOP.2.1 47

OOP.2.2 48

OOP.2.3 49

OOP.2.4 50

OOP.2.5 51

OOP.2.6 52

OOP.2.7 53

OOP.2.8 54

OOP.2.9 55

OOP.2.10 56

OOP.2.11 57

OOP.2.12 58

OOP.2.13 59

OOP.2.14 60

OOP.2.15 61

Introduction

Once we get to the point where our models consist of several classes, there will inevitably be some kind of relationship between the classes. We have already seen examples of classes that themselves refer to other classes: A **Car** class may have a relation to a **Wheel** class, an **Employee** class may have a relation to a **Teacher** class, and so on. We usually make a distinction between two fundamental kinds of class relationships: the ***has-a*** relationship, and the ***is-a*** relationship.

The Has-a relationship (Composition)

The **Car**-**Wheel** example above is a classic example of a ***has-a*** relationship. A **Car** is probably a (model of a) quite complex system, so it will make perfect sense to divide the system into a number of smaller sub-systems, each represented by a **class**. The **Car** class will thus be "composed" by several sub-classes; this sort of relationship is therefore denoted **composition** (or **aggregation**, depending on how strong the relation between the classes is). In a **Car** class definition, the relation to the other classes will be implemented by a number of instance fields, which will have the type of the classes representing the smaller systems, like

```
public class Car
{
	private Wheel[] wheels;

	public Car()
	{
		wheels = new Wheel[4];
		// (rest of Car constructor)
	}
}
```

We have seen similar examples before, and there is as such not that much more to explain about this relationship. The higher-level classes will use the lower level-classes to implement their own functionality, by using properties, calling methods, etc.. Still, the fact that classes can make use of other classes enables us to construct complex systems of collaborating classes.

The Is-a relationship (Inheritance)

A different kind of relationship can emerge as a result of discovering similarities between classes. Imagine we are working with a system for school administration. At some point, we see that we have a class **Teacher**, and another class **Secretary**. Upon further examination, we also see that the two classes have a lot in common. They probably both have properties like **Name**, **Address**, **Salary**, etc.. In accordance with the DRY principle, this is a situation we should do something about. But what? Should we merge the two classes into one larger class, that can accommodate both teachers and secretaries? That is not a good solution, since classes should be focused on a single responsibility. A different approach is to move the common parts of the two classes into a new class, and only retain the truly teacher-specific and secretary-specific parts in the original classes. The common class could be called **Employee,** and should only contain those parts common for all types of employees.

This “decomposition” of the original classes has at least brought us in line with the DRY principle, since no code is present twice anymore. Still, we need to have classes that can fully represent a teacher and a secretary, respectively. Could we then make these classes refer to the new **Employee** class through composition? That is indeed a possible solution, but it would be a somewhat convoluted version of a ***has-a*** relationship: A **Teacher** has-a **Employee**… This sounds much more like an ***is-a*** relationship: A **Teacher** is-a **Employee**. An ***is-a*** relationship is implemented by using **inheritance**.

Next-level parameterization

An ongoing theme in our progression is the idea of **parameterization**. Whenever we see an opportunity to convert a hard-coded element in a method or class into a parameter, we should seize it. Through parameterization, we enable the client (i.e. a programmer using our method or class) to choose a specific value (to be understood in a very general sense) for the parameter, instead of locking that value inside our code. As we will see below, we can take this idea even further, when we start to consider types and even code itself as potential candidates for parameterization.

Inheritance

The two classes that are part of an **is-a** relationship each play a different role. One class is more general, while the other is more specific. In the example, the **Employee** class is general, while the **Teacher** class is more specific. By “more specific”, we mean that the **Teacher** class should be everything that the **Employee** class is, plus a bit extra (the parts that are specific for a teacher). We achieve this by using **inheritance**. In C#, inheritance is syntactically quite simple to express. If we want the **Teacher** class to “inherit” from the **Employee** class, it will look like:

```
public class Teacher : Employee
{
	// Teacher-specific parts
}
```

This reads “class **Teacher** inherits from class **Employee**”. In this way, a **Teacher** object will expose all the public properties and methods that are defined in the **Employee** class, plus its own public properties and methods. To the outside world, a **Teacher** object thus appears exactly like before the “decomposition”, while we have achieved our goal of not repeating code in classes.

In more general terms, we refer to the class from which someone inherits as the **base class**, while the class that inherits (from the base class) is called the **derived class**. Another terminology is **superclass** (the base class) and **subclass** (the derived class).

The protected access level

We mentioned above that the derived class will expose all public parts of the base class, which implies that the derived class can also make use of these properties and methods inside its own properties and methods. But what about elements marked **private** in the base class? A derived class can not access these elements, so private really means private! This might seem overly strict, since we could easily imagine that a derived class would need access to certain parts in the base class, that are not available to the outside world. For this reason, there is a third access specifier named **protected**. An element in the base class marked as **protected** can indeed be accessed by a derived class, but not by an outside client.

Should we then preferably mark all elements in a class as **protected** rather than **private**, to accommodate any class that wants to inherit from the class? No. We should still be careful about what we expose, even to a derived class. If we allow a derived class direct access to all elements in the base class, it may short-circuit all sorts of validations, etc. that have been put in place in the base class. Also, the derived class might become too dependent on the internal structure in the base class, meaning that it can become difficult to change this structure if needed. We are not saying that you should never use the **protected** access level, just that you should consider the potential consequences very carefully first.

Constructors in derived classes

Even though inheritance in itself is very simple with regards to syntax, there are a number of non-trivial aspects you need to be able to consider. The first arises with regards to constructors. Imagine that we found that our **Employee** class will need a constructor with two parameters:

```
public class Employee
{
      public Employee(string name, string address)
      {
            // (rest of Employee constructor)
      }
}
```

Also, we found that the derived class **Teacher** (derived from **Employee**) only needs one parameter in its own constructor:

```
public class Teacher : Employee
{
      public Teacher(string mainSubject)
      {
            // (rest of Teacher constructor)
      }
}
```

This all looks nice here in print, but in Visual Studio, the code above will produce an error. Visual Studio will complain that *"Base class **Employee** does not contain a parameterless constructor.."*. The problem is: If you derive from a class with a parameterized constructor, you need to explicitly call this constructor when calling the constructor for the derived class, providing it with the arguments it needs. Syntactically, it will look like this:

```
public class Teacher : Employee
{
      public Teacher(string mainSubject)
        : base(name, address)
      {
            // (rest of Teacher constructor)
      }
}
```

Note the keyword **base** (preceded by a colon); this is the call to the base class constructor. In the code above, we are still not done. The parameters **name** and **address** are in red, because they are not defined anywhere… Where should they come from? The most obvious solution is that they must also be provided as part of the parameter list for the constructor of the derived class:

```
public class Teacher : Employee
{
      public string MainSubject { get; set; }

      public Teacher(string name, string address, string mainSubject)
        : base(name, address)
      {
            MainSubject = mainSubject;
      }
}
```

We have here assumed that **Teacher** contains a property **MainSubject**. This is a very common derived class constructor: the parts belonging to the base class are used for calling the base class constructor, while the parts belonging to the derived class are saved in the properties (instance fields) of the derived class itself.

Overriding methods

The inheritance mechanism described so far is actually sufficient in many scenarios. The parts defined in the base class and derived class will co-exist peacefully in objects of the derived class type, and an external user of the class will not really notice that inheritance is in play. It is however possible to use inheritance for more refined purposes, in particular to achieve so-called **polymorphic behavior**, which we describe in some detail later. A prerequisite for polymorphic behavior is the ability to "override" methods defined in the base class.

Suppose that the **Employee** class from before has a **CalculateSalary** method, which has a generic implementation sufficient for most employees. However, it turns out that salary calculation for a teacher is more complex. We must therefore implement a different way of calculating salary in the **Teacher** class. The straightforward way to do this would be to define a new method in **Teacher** called **CalculateTeacherSalary**, since we cannot call it **CalculateSalary** (as it would "collide" with the name for the method in the base class). This could be a feasible solution in many scenarios, but it is in conflict with the polymorphic behavior mentioned before.

For this reason – which will become clearer soon – we need to be able to implement a method called **CalculateSalary** in the **Teacher** class, containing the teacher-specific logic for calculating salary. This method should "override" the implementation of **CalculateSalary** in the base class, such that if you call **CalculateSalary** on an object of type **Teacher**, the teacher-specific salary calculation is executed. Is this different from before? Again, if you know you are dealing with a **Teacher** object, you might as well call a method called **CalculateTeacherSalary**? One of the main points of polymorphic behavior is however that you can call **CalculateSalary** on an object that seems like an **Employee** object, but really is a **Teacher** object, and still have the teacher-specific version of **CalculateSalary** executed! That is essentially what polymorphic behavior is.

In order to achieve this, we must however state our intention very explicitly in C#. We need to state two things:

1. In the base class, we must explicitly mark any method that may be overrided in a derived class.
2. In the derived class, we must explicitly mark any method that is overriding a corresponding method in the base class.

By "corresponding" we mean a method with exactly the same signature, i.e. same name, return type and parameter list. If just one of these don't match, we are not overriding a base class method. How does this look with regards to C# syntax? In the base class, we can state that a method may be overrided in a derived class by adding the **virtual** keyword:

```
public class Employee
{
	// (rest of Employee class definition)

	public virtual int CalculateSalary()
	{
		// Generic salary calculation
	}
}
```

Note that we only state that the method may be overrided; the derived class has no obligation to do so. If the derived class chooses to do so, it states this intention by adding the **override** keyword in its own definition of the method:

```
public class Teacher : Employee
{
      // (rest of Teacher class definition)

      public override int CalculateSalary()
      {
            // Teacher-specific salary calculation
      }
}
```

With this setup in place, we can now achieve this enigmatic "polymorphic behavior". So, what is it?

Polymorphic behavior

When we are using classes related by inheritance, we can suddenly loosen up one of our most fundamental assumptions: When you create a variable and assign a value to it, the variable and the value must have the same type. We have seen this almost from the beginning:

```
      int age = 23;

      Teacher theTeacher = new Teacher("Per", "Home", "Programming");
```

However, if **Teacher** inherits from **Employee**, the below code is also valid:

```
      Employee theEmployee = new Teacher("Ole", "Away", "Design");
```

The variable has type **Employee**, but the value has type **Teacher**... but since **Teacher** is-a **Employee** (that's what inheritance expresses), the above is also valid. But is it also useful? On its own, not so much. We have in fact restricted ourselves a bit in this way. Suppose the **Teacher** class has a property **MainSubject**. Consider then the two lines of code below:

```
      Console.WriteLine(theTeacher.MainSubject); // OK
      Console.WriteLine(theEmployee.MainSubject); // ERROR!
```

We can only use the **MainSubject** property on the variable of type **Teacher**, not on the variable of type **Employee**. This makes good sense, since that property is indeed teacher-specific. Now consider the below code:

```
Console.WriteLine(theTeacher.CalculateSalary()); // OK
Console.WriteLine(theEmployee.CalculateSalary()); // OK
```

This also makes sense, since both classes now have an implementation of **Calculate-Salary**. The big question is now:

What implementation of **CalculateSalary** will be called in each case?

The first case is probably most obvious; the variable has type **Teacher**, so the implementation in **Teacher** should be called. That is indeed true. In the second case, it is still the implementation in **Teacher** that gets called! This seems surprising, since we make the call on a variable of type **Employee**, and **Employee** has its own implementation of **CalculateSalary**. However, the C# compiler has noticed our intention of overriding the method in the derived class, and will therefore call the implementation in the derived class on any object of that type, even if the object is referred to by a variable of the base type! That is **polymorphic behavior**.

It is understandable if you still cannot appreciate why this is such a useful construct. Suppose we have a more complex system with many types of employees – all inheriting from **Employee** – where some choose to override **CalculateSalary**, while other just go with the generic implementation in the base class. Suppose also that part of the system deals with processing salaries for all employees. We could then imagine functionality like "for all employees, calculate the salary and print a salary specification". In other words, we need to iterate through all employee objects, and invoke calls to **CalculateSalary** on each object. If we had to do this without using inheritance and polymorphic behavior, we would have to maintain a list for each type of employee, in order to call the correct implementation of **CalculateSalary**, like

```
List<Teacher> allTeachers= new List<Teacher>();
allTeachers.Add(new Teacher("Hans", "Home", "English"));

List<Secretary> allSecretaries = new List<Secretary>();
allSecretaries.Add(new Secretary("Leon", "Office", "Law"));
// ..and so on

foreach (Teacher t in allTeachers)
{
       t.CalculateSalary(); // Teacher-specific salary calculation
}

foreach (Secretary s in allSecretaries)
{
       s.CalculateSalary(); // Secretary-specific salary calculation
}
// ..and so on
```

This is definitely an implementation and maintenance nightmare. With inheritance and polymorphic behavior, we can however achieve our goal with just one list:

```
List<Employee> allEmployees = new List<Employee>();
allEmployees.Add(new Teacher("Per", "Home", "Programming"));
allEmployees.Add(new Secretary("James", "Office", "Marketing"));

foreach (Employee e in allEmployees)
{
      e.CalculateSalary(); // Calls the correct implementation!
}
```

Through polymorphic behavior, we will always call the correct implementation of **CalculateSalary**, be it the generic or specific version. The above loop will not even need to be updated, if we later on add additional employee types, as long as they inherit from **Employee**. This enables much more clean and generic programming, and is definitely yet another tool for adhering to the DRY principle.

Calling base class methods

When we override methods, we often wish to replace the base class method implementation completely. However, there are also scenarios where we wish to extend the base class implementation. That is, we still want the code in the base class method to be executed, but we want to do something additional in the derived class. This could very well be the case for salary calculation: some generic parts of the calculation are done in the base class, while some more specific parts are done in the derived class. The two parts are then combined in the derived method. In the derived class, we can achieve this using the following syntax:

```
public override int CalculateSalary()
{
      return base.CalculateSalary() + payGrade*500;
}
```

Again, the keyword **base** is used to refer to a base class implementation. The specific way to combine the result of calling the base class implementation and the result of the derived class implementation will of course vary from case to case.

Abstract methods (and classes)

One of our assumptions in the above example was that some sort of common salary calculation logic exists, that can be used if no extra salary calculation logic applies for a specific kind of employee. What if the salary calculation logic is so diverse that no common logic exists? What will the implementation of **CalculateSalary** in **Employee** then look like? Maybe this:

```
public virtual int CalculateSalary()
{
	return 0;
}
```

This could be the case, maybe with the argument *"Well, we have to put something, right?"*. That is however not a valid argument. In general, we often face this situation:

- All classes inheriting from a base class **B** should implement a method **M**
- The is no meaningful implementation of **M** in **B** itself

First of all, what is the problem with the not-so-meaningful implementation of **CalculateSalary** above? We will override the method in all derived classes anyway, yes? True, if we remember to do it! There will be nothing alerting us that we have forgotten to override it for a new derived class, except that some employee might see a zero on his salary specification… It would be much better if we could make it mandatory for all classes inheriting from **Employee** to implement **CalculateSalary**, but without having a meaningless default implementation in **Employee**. We can achieve this by making the **CalculateSalary** method **abstract**.

An abstract method is a method without a body… that is, we only specify the method signature in the class definition:

```
public abstract class Employee
{
	public abstract int CalculateSalary();

	// (rest of Employee class)
}
```

Notice the semi-colon at the end. We are really done with what we have to say about this method in the **Employee** class. We only specify its signature, nothing more. If a class inherits from **Employee**, it will now be required to implement **CalculateSalary**.

Declaring an abstract method in a class has an additional consequence. Consider the below (invalid) code

```
Employee e = new Employee("Vivian","Home"); // ERROR!
e.CalculateSalary(); // What should happen here?
```

What should indeed happen in the second line? It's meaningless, since there is no implementation of **CalculateSalary** to call. In general, any class that contains an abstract method will itself need to be marked as **abstract** (as **Employee** is above), implying that you can not create an object of that type! However, you can still have a variable of that type, so the ability for polymorphic behavior is preserved.

With this in place, we can sum up the difference between a **virtual** method and an **abstract** method:

- **Virtual** method: Has an implementation in the base class, can be overridden in a derived class. Use when a meaningful implementation of the method can be provided in the base class
- **Abstract** method: Does not have an implementation in the base class, must be overridden in a derived class. Use when no meaningful implementation of the method can be provided in the base class

Interfaces

The concept of defining abstract methods in a class can be taken to the extreme; a class that only contains abstract methods. This is actually a very useful idea, and has even been given its own name in Object-Oriented programming: an **interface**.

An interface can be seen as the absolutely minimal specification of a class. Imagine somebody interested in some particular functionality, for instance a class capable of drawing geometric shapes. How can he state his requirements in a very precise way, but without any assumption about specific implementation details? In terms of an interface definition!

An interface definition for a (very simple) geometry drawing system could be specified in C# as:

```
public interface IGeometryDraw
{
      void DrawCircle(double x, double y, double radius);
      void DrawLine(double x1, double y1, double x2, double y2);
      void DrawRectangle(double x1, double y1, double x2, double y2);
}
```

There are several things to take note of here:

- The keyword **interface** is used instead of **class**
- The interface name starts with an I – this is a naming convention
- There is no access specifier – all methods are per definition **public**
- There is no constructor
- All methods are by definition **abstract**

Somebody interested in obtaining this functionality could simply state it in terms of this interface definition, alongside a specification of what an implementation of the interface is supposed to do, from an external perspective. It will of course not make sense to implement the **DrawCircle** method in a way that draws a square, so some sort of requirement specification or "contract" is needed. But apart from that, there is no need for additional information. A programmer could then go back and create a class that implements the interface:

```
public class GeometryDrawV10 : IGeometryDraw
{
      public void DrawCircle(double x, double y, double radius)
      {
            // (rest of DrawCircle)
      }

      // (rest of GeometryDrawV10)
}
```

The syntax for implementing an interface is identical to the syntax for inheritance in general. One important difference is however that a class can "inherit" from (i.e. implement) multiple interface, but can only inherit from a single non-interface class. The reasons for this limitation are a bit technical, and beyond the scope of this text.

If you take a tour through parts of the .NET class library, you will see that interfaces are used quite heavily. Use of interfaces is a very strong mechanism for making couplings between classes as weak as possible, since they are a specification of the absolute minimum you need to know about a class in order to use it. We will use interfaces quite often in the rest of these notes.

The Object class

We conclude this section on inheritance by a small revelation – we have been using inheritance all along. All C# classes tacitly inherit from a "universal" base class named **Object**. This base class has seven methods:

- **GetType**
- **Equals**
- **GetHashCode**
- **ToString**
- Finalize
- MemberwiseClone
- ReferenceEquals

Some of these methods (**Equals**, **GetHashCode** and **ToString**) can be overrided in a class definition, if you want your class to have certain abilities. A particularly interesting method is the **ToString** method. As we have seen many times before, we can put anything into a **Console.WriteLine** method call; the method will then try to print out the argument. This works well for simple types like e.g. **int**, where the value is printed as expected. However, if we try to do something like this:

```
Console.WriteLine(theTeacher);
```

we will get something like **MyNamespace.Teacher** printed on the screen. What happens is that the method tries to print the string representation of the argument, more specifically by calling the **ToString** method – which all classes implement due to the inheritance from **Object** – and print the return value. What we see is the base class implementation of **ToString**. If we want a more useful result, we can override the **ToString** method in the **Teacher** class:

```
public override string ToString()
{
      return Name + " teaches " + MainSubject;
}
```

Now we will see a printout like e.g. "John teaches Design", whenever the program tries to print a **Teacher** object.Some of the remaining **Object** methods can also be overrided for more or less exotic purposes; seek up additional information online about this, if you find that you need to do so.

Exceptions

So far, we have given very little consideration to how to handle error situations. Handling error situations is usually a pretty significant issue in software development, so we cannot ignore it; we need to know about tools and strategies for managing it.

So, what can go wrong? Below are just a few of the error situations we can imagine for a simple value:

- Value is correct type-wise, but is outside the range of meaningful values (example: A test score is supposed to be between 0 and 100, but an **int** can represent many other values, like e.g. -27, 22987).
- Value is used for indexing an array – only values from 0 (zero) up to (**Length** - 1) are meaningful. Other values will produce an error.
- A string does not follow a given syntax (e.g. for a license plate).
- A variable that is supposed to refer to an object has the value **null** instead.

The proper action of the application in the above cases will be situation-dependent. The application may halt, show an error message, silently handle the error, fall back to a default value, etc. In any case, we should be prepared for handling all possible error situations in a graceful manner. Simply shutting down the application is usually not an acceptable option.

Management of error situations can in general be divided into four phases:

1. **Detection** – realizing that an error situation has occurred
2. **Signaling** – making the surrounding code aware that an error has been detected
3. **Capturing** – taking responsibility for handling the error
4. **Handling** – actually performing the error handling actions

The actions corresponding to these phases can be distributed in the code. This may imply that the part of the code detecting the error does not know how to handle the error! Information about the error must then somehow be propagated to the error handling code. One way of doing this could be to use return values. A method could return some sort of error code or object as its return value, which the caller could then act upon. This strategy does however quickly turn out to become very complicated, so we usually resort to a different mechanism: **exceptions**.

Throwing and catching

Exceptions are by themselves just a set of classes, that all inherit from the .NET library class **Exception**. If you need to use an exception object, you create it using **new**, just as for any other object. The distinctive feature for exceptions is that you can “throw” and “catch” exception objects.

If an error situation – or “exceptional” situation, to use a broader term which also includes errors – occurs, the code which **detects** the situation can “throw” an exception. In C# code, this will look like:

```
public void Deposit(int amount)
{
      if (amount < 0) // Error detected
      {
            NegativeAmountException ex = new NegativeAmountException("Deposit");
            throw ex;
      }

      _balance = _balance + amount;
}
```

Here **NegativeAmountException** is a class we have defined ourselves. It inherits from **Exception**, which makes a **NegativeAmountException** object “throwable”. Note the keyword **throw** – this is where the object gets thrown. This is the **signaling** phase of the error handling process. When an exception object is thrown, no more code in the method is executed, just as if we had used the **return** keyword.

What does it mean more specifically to “throw” an object? Throwing an object is different from returning an object. An object which is thrown is passed up through the method calling chain just as return values are, but – and this is a significant difference – a method that has no interest in exceptions does not need to do anything at all in terms of handling it. A thrown object will silently pass up through the method calling chain, until someone decides to “catch” the exception object.

Catching an exception object is something a caller deliberately chooses to do. If you are calling a method that might throw an exception, and you are intent on handling the exception if it occurs, you should encapsulate the call in a **try-catch** statement:

```
BankAccount theAccount = new BankAccount();
try
{
	theAccount.Deposit(-1000);
}
catch (NegativeAmountException ex)
{
	Console.WriteLine($"{ex.Message}: Negative amount not allowed");
}
```

This may look a bit tedious with regards to syntax, but remember that it is only if you have the intent of actually handling the error, that you need to use this construction.

How do we read the above code? The caller is aware that **Deposit** might throw an exception, so he places the call in the **try**-part of the statement. If the call goes well, nothing else happens – the **catch**-part does not come into play. However, if an exception is thrown, the exception is caught by the **catch**-part (this is the **capturing** phase). Now the code in the **catch**-part is executed. In this simplified case, the code does a very simplistic error handling (this is the **handling** phase), by printing out a message. In a more realistic setting, the code might make a call to some code dedicated to error handling, error recovery and error presentation. Once the error handling code finishes, the statements following the entire **try-catch** statement will be executed.

We said above that the **catch**-part was executed if the **Deposit**-statement threw an exception. That is not entirely accurate. The code will only be executed if an exception object of the type **NegativeAmountException** is thrown. A **catch**-statement will only catch exceptions of that type we have specified in the parentheses following the **catch** keyword. If a different exception had been thrown, it would not have been caught here, but possibly further up the method calling chain. If we want to catch more than one type of exceptions, we can simply write additional **catch**-blocks after the first one, like:

```
BankAccount theAccount = new BankAccount();
try
{
	theAccount.Deposit(-1000);
}
catch (NegativeAmountException ex)
{
	Console.WriteLine($"{ex.Message}: Negative amount not allowed");
}
catch (LargeAmountException ex)
{
	Console.WriteLine($"{ex.Message}: Amount must not exceed …");
}
```

In the above example, it seems reasonable to assume that **NegativeAmount-Exception** and **LargeAmountException** are two exception classes defined on the same level in an exception class inheritance hierarchy. They might both inherit from the .Net library class **ArgumentException** (which itself inherits from **Exception**). Suppose now that your exception handling logic is as follows:

- **LargeAmountException** (which inherits from **ArgumentException**) is handled according to strategy A
- **ArgumentException** (other than **LargeAmountException**) is handled according to strategy B
- **Exception** (other than **ArgumentException**)is handled according to strategy C

Can we express this in C# in a simple way? Indeed we can, again by including multiple **catch**-blocks in our **try-catch** statement:

```
try
{
      // Some action which may generate an exception
}
catch (LargeAmountException ex)
{
      // Strategy A
}
catch (ArgumentException ex)
{
      // Strategy B
}
catch (Exception ex)
{
      // Strategy C
}
```

In this way, you specify the most “specialized” error handling first, followed by more and more general error handling. When an actual exception is generated, the application will execute the first **catch**-clause (and only that clause) which matches the type of the exception.

Rethrowing an exception

What if you want to do something if an exception occurs, but also want others to have a chance of handling the exception? You can then “re-throw” the exception. A common scenario could be that you wish to do some sort of logging of the exception, but also want to do more specific exception handling further up the call chain. You can then do a re-throw in the style illustrated below:

```
try
{
      // Code that may throw exceptions
}
catch (Exception ex)
{
      Logger.Log(ex.Message); // Log exception
      throw;  // Rethrow exception
}
```

This is a quite useful construction, where someone having a stake in error handling can perform a specific kind of handling, but also pass on the exception to others.

Exceptions summary

We now know the essentials of dealing with exceptional situations using exception objects. The most important points are:

- The .NET class library contains a lot of exception classes, including the class **Exception**, which is the base class for all exception classes. If you need a specialized exception class, explore the library first. There might be a class that fits your needs.
- You can define your own exception classes, but they must inherit from **Exception**, or one of the other existing exception classes (including exception classes you have defined yourself).
- You should **throw early**: as soon as you have discovered an error situation that you don't want to handle yourself, throw an appropriate exception.
- You should **catch late**: do not catch an exception unless you are sure what to do with it. Do not catch an exception and then ignore it by doing nothing.
- Consider **rethrowing** the exception, when you have dealt with it. Others might want a chance to handle the exception as well.

Exceptions is a construction that is a bit contrary to the ordinary flow-of-execution, but once you get a grasp of it, it is a quite elegant and powerful way of dealing with errors in a non-intrusive way. Remember, you only need to deal explicitly with (i.e. write code for) exceptions, if you have an interest in handling them.

Generics – types as parameters

We discussed the DRY (Don't Repeat Yourself) principle a while ago, and considered the principle at various levels (instance field, method and class level). The goal was always the same: avoid writing the same code over and over. At the class level, we saw that **inheritance** is a useful mechanism for avoiding code duplication, since you can place code shared by several classes in a base class, which can then be inherited from. Still, some situations are not easily solved by inheritance.

Shortcomings of inheritance

Suppose we have defined a simple domain class **Dog**, and also wish to have a way of representing family relations between dogs. Instead of defining family relations as a part of the **Dog** class itself, we decide to create a separate class **FamilyRelation**, which will represent family relations between **Dog** objects. Such a class could look like this:

```
public class FamilyRelation
{
	private Dog _self;
	private Dog _father;
	private Dog _mother;
	private List<Dog> _children;

	public FamilyRelation(Dog self, Dog father, Dog mother)
	{
		_self = self;
		_father = father;
		_mother = mother;
		_children = new List<Dog>();
	}

	public Dog Self { get { return _self; } }

	public Dog Father { get { return _father; } }

	public Dog Mother { get { return _mother; } }

	public List<Dog> Children { get { return _children; } }

	public void AddChild(Dog child)
	{
		_children.Add(child);
	}
}
```

This is pretty straightforward, and we can easily start to use this class:

```
	Dog self = new Dog("King");
	Dog mom = new Dog("Spot");
	Dog dad = new Dog("Rufus");

	FamilyRelation relations = new FamilyRelation(self, mom, dad);
	relations.AddChild(new Dog("Lajka"));
```

So far, so good. Now we also define a domain class **Cat**, and also wish to be able to represent family relations between **Cat** objects. We cannot use the **FamilyRelation** class as-is, since it operates on **Dog** objects. We can do a copy-paste of the class, and create a (very similar) class **FamilyRelationCats**, where we simply replace **Dog** with **Cat**. However, this is exactly the situation we wish to avoid…

Cat and **Dog** are probably strongly related classes, and it will probably make sense to define a base class **Animal**, from which both **Dog** and **Cat** can inherit. If we do that, we could also update the **FamilyRelation** class:

```
public class FamilyRelation
{
	private Animal _self;
	private Animal _father;
	private Animal _mother;
	private List<Animal> _children;

	// ...and so on
}
```

Now we can use the **FamilyRelation** class for both **Dog** and **Cat** objects. There are however several problems with the class:

1. Nothing will prevent us from mixing **Cat** and **Dog** objects, so a cat could e.g. be the father of a dog…
2. The return type of the properties will be **Animal**, so we will need to try to cast the returned object to a derived class, if we need to do something **Cat**- or **Dog**-specific with the object.
3. We can only use the **FamilyRelation** class for classes inheriting from **Animal**, even though the **FamilyRelation** class itself does not use any **Animal**-specific methods or properties.

Using inheritance cannot really solve these problems for us. What we really want is to turn the **type** of the objects used in **FamilyRelation** into a parameter. That is, we wish to define the **FamilyRelation** class with a general type parameter, and wish to use the **FamilyRelation** class with a specific type as argument. This is essentially the same strategy we use when defining a simple method; we define the method in a general way, possibly by including parameters:

```
public int ReturnLargest(int a, int b)
{
      return (a < b ? b : a);
}
```

and we use the method with specific values as arguments:

```
ReturnLargest(7, 12);
```

Using types as parameters

The feature in C# called **Generics** is exactly the ability to use types as parameters to class definitions (and to methods, which we will also see examples of). We have used this ability already without calling it Generics, when we saw examples of **data structures**. If we needed a list of integers, we could declare it like this:

```
List<int> myNumbers = new List<int>();
```

This will create a **List** object, into which we can only insert **int** values. Also, any method that returns an item in the list will have the return type **int**. In that sense, the **List** class – or more correctly, the **List<T>** class – is **type-safe**. We cannot accidentally insert elements of different types into the list, and the items returned from the list have the correct type, i.e. no need for casting. The **<T>** following the **List** class name indicates that the **List** class takes one type parameter – just as a method can take one parameter – which must be specified when using the class, as above. Type parameters can be called whatever you like – just as a parameter to a method – but are usually called **T** (T for Type, maybe…).

With this knowledge, we can create a new and more generally applicable version of the **FamilyRelation** class:

```
public class FamilyRelation<T>
{
      private T _self;
      private T _father;
      private T _mother;
      private List<T> _children;

      public FamilyRelation(T self, T father, T mother)
      {
            _self = self;
            _father = father;
            _mother = mother;
            _children = new List<T>();
      }

      public T Self { get { return _self; } }

      public T Father { get { return _father; } }

      public T Mother { get { return _mother; } }

      public List<T> Children { get { return _children; } }

      public void AddChild(T child)
      {
            _children.Add(child);
      }
}
```

We have simply substituted **Dog** with **T**, and added the **<T>** type parameter declaration after the **FamilyRelation** class name. At first sight, this may look confusing. What is **T** actually? Is it a new class? No, it is simply a (type) parameter to the class. Again, think of it as very similar to a parameter to a method. You can give it any name you like, and when you use the method, you must provide a specific value as argument. If we now wish to use the **FamilyRelation** class, we must provide a specific type:

```
      FamilyRelation<Dog> relations = new FamilyRelation<Dog>(self, mom, dad);
      relations.AddChild(new Dog("Lajka", 45, 20));
```

The variable **relations** now refers to a **FamilyRelation** object, specialized to the **Dog** type. This solves the problems listed before:

1. We can only insert **Dog** objects into the **FamilyRelation** object
2. All properties have the return type **Dog** (or **List<Dog>**)
3. **Dog** does not need to inherit from a certain base class in order to be used with the **FamilyRelation** class.

Using the **FamilyRelation** class to define relations between **Cat** objects is now as easy as it gets:

```
Cat a = new Cat("Stripe");
Cat b = new Cat("Socks");
Cat c = new Cat("Abby");

FamilyRelation<Cat> catRelations = new FamilyRelation<Cat>(a, b, c);
```

It doesn't take much consideration to conclude that <u>any</u> type – even simple types like **int** – can be used with **FamilyRelation**. Using a class called **FamilyRelation** to represent relations between numbers is perhaps a bit warped, but the point is still valid. This could also be seen as an argument for choosing the non-descriptive name **T** for the type parameter. We could have chosen to call the type parameter **TAnimal** (not to be confused with a base class called **Animal**) to try to indicate the intended use of the class, but that would not prevent anyone from using it with a completely unrelated type like **int**. The name **T** has no implicit meaning in itself, and thus indicates that "anything goes" with regards to choice of type.

Type constraints

The above considerations about what types to use with **FamilyRelation** leads directly to an important Generics sub-topic: **type constraints**.

The **FamilyRelation** class could be type-parameterized very easily, since the class does little more than store and return items of type **T**. The term "item" is chosen deliberately, since these items may not even be objects, if a simple type like **int** is used. If we begin to add functionality involving the stored items, we may however run into problems quite soon. Suppose we add the modest requirement that we can create a new **FamilyRelation** object, when only the "self" is known. We suppose that the mother and/or father can be added later. This could be done by adding an extra constructor, like this:

```
public FamilyRelation(T self)
{
        _self = self;
        _father = null;
        _mother = null;
}
```

Trying to compile this code will produce an error *“Cannot convert **null** to type parameter **T**, because it could be a value type”*. This seems reasonable; if the chosen type is **int**, we are trying to assign **null** to an instance field of type **int**, which does not make much sense.

This could be an opportune moment to consider, if we really want to allow use of the **FamilyRelation** class with any type. If we come to the conclusion that the used types should at least be class types (i.e. not simple types like **int** or **bool**), we can specify this by adding a **type constraint** to the class definition:

```
public class FamilyRelation<T> where T : class
{
	// Rest of class definition omitted
}
```

This expresses that **T** must at least be of a class type. Adding this constraint to the class definition has two consequences in relation to the previous code:

1. The extra constructor is now valid and can compile.
2. Using **FamilyRelation** with type **int** now causes a compilation error.

This is exactly as intended. We can take this principle further, and e.g. constrain **T** to be a class that inherits from a base class **Animal**:

```
public class FamilyRelation<T> where T : Animal
{
	// Rest of class definition omitted
}
```

Placing this constraint on **T** also allows us to start using the instance fields, i.e. call methods on the objects referred to by the instance fields. If the **Animal** class contains a property **Name**, the below will be legal code inside the **FamilyRelation** class:

```
	public void PrintNamesOfParents()
	{
		Console.WriteLine($"Parents: {_father.Name} and {_mother.Name}");
	}
```

Due to the constraint on **T**, we are now guaranteed that the **Name** property will always be available, and it is therefore valid to use it.

Deciding exactly how to constrain a type parameter can be a bit tricky. It will always be a balance between keeping the class as general as possible (i.e. keeping the constraints minimal) and being able to perform type-specific operations within the class (i.e. having enough constraints). You should generally strive to have "just enough constraints". If a constraint can be removed without causing compilation errors, it should obviously be removed. You should also consider carefully what operations you really need to perform with type-parameterized objects. Finally, Visual Studio is often able to provide useful suggestions for adding (or removing) constraints on type parameters.

Type parameter variance

A very natural question relating to type parameters concerns how they relate to inheritance. More specifically: If we have defined a class **A**, and also define a class **B** which inherits from **A**, we know from Polymorphism whether or not the below lines of code are valid:

```
A a = new B(); // OK
B b = new A(); // Error
```

Suppose now we have another class **C<T>**, which takes one type parameter **T**. This could be the **FamilyRelation** class defined above. What would we except about these lines of code?:

```
C<A> ca = new C<B>(); // ??
C<B> cb = new C<A>(); // ??
```

Based on intuition, many will probably guess that the same validity applies here, such that the first line is valid while the second is not. It turns out that <u>none</u> of these lines of code are valid… The reasons for this are a bit tricky, but let's have a look at it.

First of all, we will use the **Animal** class as a base class, and the **Dog** class as subclass. We thus have this code as our starting point:

```
Animal a = new Dog("Spot"); // OK
C<Animal> ca = new C<Dog>(); // Error
```

The class **C** is now defined as having a single instance field and two very simple methods:

```
public class C<T>
{
	private T _t;

	public T Get()
	{
		return _t;
	}

	public void Set(T t)
	{
		_t = t;
	}
}
```

Let us now for a moment assume that the below line is in fact valid:

```
	C<Animal> ca = new C<Dog>();
```

This will create an object of type **C**, where **T** is set to **Dog**. So, what type of object can be returned by the call **ca.Get()**? Only an object of type **Dog**, and since **Dog** is a sub-class of **Animal**, this is a perfectly valid object to return in response to calling the **Get** method on **ca**, since this method has the return type **Animal**. Remember that **C<Dog>** does not inherit from **C<Animal>**, so this is not a matter of polymorphism! The conclusion is thus that with respect to the **Get** method, the above statement would be safe.

What about the **Set** method? If we call **ca.Set(…)**, what type of objects will then be valid arguments to **Set**? Since **ca** has the type **C<Animal>**, the **Set** method will accept any object of type **Animal**, including – but not limited to – **Dog**. The last part is what breaks our assumption. If e.g. **Cat** inherits from **Animal**, we can call **ca.Set(…)** with a **Cat** object, which cannot be inserted into an object of type **C<Dog>**…

Let's also consider the second line, which does look very contra-intuitive:

```
	C<Dog> cd = new C<Animal>();
```

We immediately run into problems when considering the **Get** method. This method can only return objects of type **Dog**, but we may easily have an object of a different type (e.g. **Cat**) in the referred-to object, which breaks our assumption. However, the **Set** method does not break the assumption! In this case, the **Set** method only accepts objects of type **Dog**, which we can safely insert into an object of type **C<Animal>**, since **Dog** inherits from **Animal**.

The conclusion so far is thus: Given our current implementation of the class **C<T>**, it makes perfect sense that the two assignment statements from above are invalid. There does however seem to be a pattern as to how the validity is broken. Methods that return a value of type **T** induce one kind of "breakage", while methods taking a parameter of type T induce a different kind. It turns out that this difference can be exploited further.

Let's now add two **interfaces** to the mix. We define two interfaces **IGet** and **ISet**, respectively:

```
      public interface IGet<T>
      {
            T Get();
      }

      public interface ISet<T>
      {
            void Set(T t);
      }
```

The original class **C<T>** now implements these two interfaces:

```
public class C<T> : IGet<T>, ISet<T>
{
      private T _t;

      public T Get()
      {
            return _t;
      }

      public void Set(T t)
      {
            _t = t;
      }
}
```

We can now try out some slightly different statements:

```
      IGet<Animal> iga = new C<Dog>();
      ISet<Dog> isd = new C<Animal>();
```

With the above class definitions, both lines of code are deemed invalid by the compiler. However, if you perform an analysis similar to the previous analysis, you come to the conclusion that both lines are actually safe, and thus in principle valid. So why does the compiler not agree? It turns out that you have to specify your "intention" with a type parameter explicitly in the interface definitions:

```
public interface IGet<out T>
{
        T Get();
}

public interface ISet<in T>
{
        void Set(T t);
}
```

The keywords **in** and **out** make our intention explicit: A type parameter marked with **in** will only be used as a type for "input" parameters to methods, while a type parameter marked with **out** will only be used as a type for return values for methods and properties. The compiler should in principle be able to deduce this, but it turns out to be a quite hard problem in practice, which is why it was decided by the designers of C# that the programmer must state this explicitly. In C#, it is also only allowed to add these keywords to type parameters for interfaces, not for classes in general.

The keywords **in** and **out** are pretty closely related to the stated intentions; to use the type parameter only for input or output, respectively. More formally, for an interface like **IGet<out T>**, **T** is said to be declared as being **co-variant**, while for an interface like **ISet<in T>**, **T** is said to be declared as being **contra-variant**. These terms originate from so-called "category theory" in Mathematics. If a type parameter is not marked with either **in** or **out**, it is said to be **invariant**. The keywords **in** and **out** are probably easier to remember… As with type constraints, the development environment is capable of suggesting when to declare a type parameter to be co- or contra-variant.

The final – and most important – question in relation to this topic is of course: *Should I care about this?* Is it just an academic observation, or does it have any practical consequences? Whether or not you will ever face a situation where type parameter variance will be a crucial matter, is of course very hard to predict. Still, you need not look further than the .NET class library for some very concrete examples. We will take a closer look at two commonly used interfaces in the library, called **IComparable<T>** and **IComparer<T>**.

The IComparable<T> and IComparer<T> interfaces

Continuing our example with the **Animal** class and the **Cat** and **Dog** subclasses, we can imagine that it at some point becomes necessary to sort a list of e.g. **Dog** objects, for instance according to weight. Since weight is a property all animals have, it makes sense to implement a **Weight** property in the **Animal** base class. Having done this, we can go ahead and try to sort a list of **Animal** objects:

```
List<Animal> animals = new List<Animal>();
animals.Add(new Dog("King", 70));
animals.Add(new Dog("Spot", 30));
animals.Add(new Dog("Rufus", 80));
animals.Sort();
```

This naïve attempt will not succeed; in fact, the code will cause an exception to be thrown when the **Sort** method is called (an **InvalidOperationException**). This is a reasonable reaction, since the **Sort** method has no way of knowing that we wish to sort according to weight. How can we state this intention to the **Sort** method? In order to be able to sort a set of objects, we must be able to compare them to each other. An object must be "greater than", "equal to" or "smaller than" another object, in order to perform a meaningful sorting of the objects. One way of achieving this is to let the objects implement the **IComparable<T>** interface. This interface contains just one method **CompareTo**:

```
int CompareTo(T other);
```

The specification of the behavior of the **CompareTo** method is as follows:

if	then return
The object on which the method is called is **smaller than** the argument object	A value **less than 0** (zero).
The object on which the method is called is **equal to** the argument object	**0** (zero).
The object on which the method is called is **greater than** the argument object	A value **greater than 0** (zero).

A valid implementation of **CompareTo** in the **Animal** class is therefore:

```
public int CompareTo(Animal other)
{
        if (Weight < other.Weight) { return -1; }

        if (Weight > other.Weight) { return 1; }

        return 0;
}
```

If we now attempt to run the code containing the call of **Sort** on the list of **Animal** objects, the code will execute without errors, and the list will be sorted as intended. How does this relate to type parameter variance? Only by the fact that the type parameter **T** in the **IComparable<T>** interface is in fact declared as being contra-variant, since **T** is only used as the type for the method parameter (i.e. input).

Letting your class implement the **IComparable<T>** interface is one way of making the objects “comparable”, and thereby sortable. However, there are two drawbacks:

1. You may not always be able to let a class implement **IComparable<T>**. The class may be a third-party class, or may due to other circumstances be “closed for modification”.
2. You lock the comparison to one specific implementation. You might need to sort the objects according to a different criterion, which would then require modification of the **CompareTo** method.

An alternative is to use the **IComparer<T>** interface. This interface is not an interface that the class itself should implement, but rather an interface that a class dedicated to comparing objects of type **T** should implement. The interface is as such very similar to the **IComparable<T>** interface:

```
int Compare(T x, T y);
```

The specification of the behavior of the **Compare** method is as follows:

if	**then return**
Object **x** is **smaller than** object **y**	A value **less than 0** (zero).
Object **x** is **equal to** object **y**	**0** (zero).
Object **x** is **greater than** object **y**	A value **greater than 0** (zero).

We can then create a new class **AnimalComparerByWeight**, like this:

```
public class AnimalComparerByWeight : IComparer<Animal>
{
      public int Compare(Animal x, Animal y)
      {
            if (x.Weight < y.Weight) { return -1;}

            if (x.Weight > y.Weight) { return 1;}

            return 0;
      }
}
```

Again, note that this is a brand new class, that is not part of the **Animal** class itself. With this class available, we can sort our list of **Animal** objects in a slightly different way:

```
      IComparer<Animal> comparer = new AnimalComparerByWeight();
      animals.Sort(comparer);
```

We have now separated the comparison functionality from the **Animal** class itself, and can now provide a specific implementation of **IComparer<Animal>** as an argument to the **Sort** method. If we later wish to compare (and sort) **Animal** objects by a different criterion, we simply create a new class containing a different implementation of **IComparer<Animal>**, and use an object of that class as an argument to **Sort**.

Is this then an illustration of contra-variance? Yes, in the sense that **T** is also declared as contra-variant for this interface. A more tangible advantage does however reveal itself, if we make a small modification to our code:

```
      List<Dog> animals = new List<Dog>();
      animals.Add(new Dog("King", 70));
      animals.Add(new Dog("Spot", 30));
      animals.Add(new Dog("Rufus", 80));

      IComparer<Animal> comparer = new AnimalComparerByWeight();
      animals.Sort(comparer);
```

The **animals** list is now a **List** of **Dog** objects, not a **List** of **Animal** objects. That is, the type parameter **T** is now **Dog**. If you read the specification of the **Sort** method (the version taking an **IComparer** implementation as a parameter), you will see that the type of the parameter is **IComparer<T>**, meaning that the actual type should now be **IComparer<Dog>**, where it previously was **IComparer<Animal>**. However, the variable **comparer** has type **IComparer<Animal>**… Still, the method does accept the argument of type **IComparer<Dog>**, because of contra-variance! What effectively happens when using **comparer** as argument is something like this:

```
IComparer<Dog> ic = comparer; // of type IComparer<Animal> !!
```

This is exactly the kind of contra-intuitive assignment we discussed earlier, which is only feasible due to the contra-variance of **T** in **IComparer<T>**. If this was not possible, what would the consequence be? You would then need to create a separate class implementing **IComparer<Dog>** – so it could be used as a parameter for **Sort** – but also a class implementing **IComparer<Cat>**, and a similar class for all subclasses to **Animal**! This would be a severe drawback as compared to having a single implementation in the **Animal** base class. So even though co- and contra-variance may appear a bit academic, it does provide tangible and significant advantages in practice.

Generic methods

A final point in relation to Generics is the fact that you can also use Generics at the method level. The classic example is a method **Swap** for swapping two values (the **ref** keyword means that the arguments are "by-reference", which means that the values of **a** and **b** will indeed by swapped, also after the method has completed. If we omitted the **ref** keyword, it would be copies of **a** and **b** that would be swapped):

```
public void Swap<T>(ref T a, ref T b)
{
      T temp = a;
      a = b;
      b = temp;
}
```

Just as for classes, you need to specify a concrete type when calling **Swap**, like this (assuming that the type of **swapper** is a class containing the **Swap** method):

```
int x = 12;
int y = 21;
swapper.Swap<int>(ref x, ref y);
```

In practice, it turns out that the compiler can often figure out the correct type by examining the type of the arguments, so you can actually omit the type specification:

```
int x = 12;
int y = 21;
swapper.Swap(ref x, ref y);
```

You can specify a generic method in any class (and interface), and the class itself does not need to have any type parameters. Finally, you can also impose constraints on the type parameter, using the same syntax as for classes:

```
public void Swap<T>(ref T a, ref T b) where T : class
{
      T temp = a;
      a = b;
      b = temp;
}
```

Functions as parameters

The previous chapter has (hopefully) illustrated that the parameter concept goes beyond simple data, since we can perceive types as parameters as well. This ability helps us define code that is as general as possible, postponing the specific choices for values and types to invocation rather than definition. The next step down this road is to perceive **functions** (i.e. methods) as potential parameters as well.

A first attempt at function parameterization

Consider for instance the problem of finding an object matching certain conditions in a collection. Suppose we have defined a simple class **Car**, like this:

```
public class Car
{
      private string _licensePlate;

      public string LicensePlate { get { return _licensePlate; } }

      // Rest of class definition omitted
}
```

We could then imagine storing **Car** objects in a **Dictionary<string, Car>**, since the **LicensePlate** property is a good candidate for acting as a key for **Car** objects. This makes it very easy to retrieve a **Car** object with a specific license plate:

```
      string licensePlate = "CJ 32 802";
      if (carsDict.ContainsKey(licensePlate))
      {
            Console.WriteLine(carsDict[licensePlate]);
      }
```

Now suppose that we for some reason – maybe due to the usage pattern for our collection of **Car** objects – have decided to store the **Car** objects in a **List<Car>** object instead. This makes it a bit harder to find a **Car** object with a specific license plate, since we have to explicitly iterate through the collection:

```
      Car theCar = null;
      foreach (Car aCar in carsList)
      {
            if (aCar.LicensePlate == licensePlate)
            {
                  theCar = aCar;
            }
      }
```

This is a very generic piece of code. If we wish to find a **Car** object matching a different condition, we only have to change the condition in the **if**-statement:

```
Car theCar = null;
foreach (Car aCar in carsList)
{
	if (aCar.Price == price)
	{
		theCar = aCar;
	}
}
```

Since we like DRY code (Don't Repeat Yourself), it would be nice if we could just write this code once, and then turn the part that varies into a parameter. This is a principle we generally apply when making methods as general as possible. However, the part that varies is now not just a simple data value or a type; it is a small piece of logic.

What is the nature of the code in the condition part of the **if**-statement? It definitely returns a **bool** value, since it is indeed used as a condition. It also seems reasonable to assume that the condition always involves a **Car** object, since the whole purpose of the code is to select a specific **Car** object. We can then think of that piece of code as being a function itself, with at least two properties:

- It takes a **Car** object as input
- It returns a **bool** value

We can compare this with the four elements we always require in order to define a proper function:

- A name
- A parameter list
- A return type
- A body

So far, we only have two of these. In order to create a proper function for e.g. the case where we want a match on the **Price** property, we could create a method named **PriceMatch**:

```
private bool PriceMatch(Car aCar, int price)
{
	return aCar.Price == price;
}
```

We can then rewrite the loop from above as:

```
Car theCar = null;
foreach (Car aCar in carsList)
{
	if (PriceMatch(aCar, price))
	{
		theCar = aCar;
	}
}
```

However, this is not really enough. If we want to match on e.g. the license plate, we will have to alter the code once again. What we really want is to turn the condition itself into a parameter, like this (**NB**: the below code does not compile):

```
public Car FindCar(List<Car> carsList, Condition matchCondition)
{
	Car theCar = null;
	foreach (Car aCar in carsList)
	{
		if (matchCondition(aCar, ...))
		{
			theCar = aCar;
		}
	}

	return theCar;
}
```

We should then be able to call this code like so:

```
FindCar(carsList, PriceMatch);
```

We're closer, but this doesn't work either. The problem is that the condition methods require different sets of arguments, depending on their specific implementation. The **PriceMatch** method requires a price (of type **int**), a **LicensePlateMatch** method may require a license plate (of type **string**), and so on. Are we then at a dead end? Fortunately not! We can solve this problem by introducing **lambda expressions**[1].

[1] The language construction we denote as **lambda expressions** can also be denoted as **anonymous functions**. There are some subtle differences between the two concepts; we will use the term lambda expression here, even though it may in some situations not be 100 % accurate.

Lambda expressions

The short definition of a lambda expression is *"an expression that returns a function"*. This may sound very abstract, but it is actually not so different from what we have seen before. We have definitely seen both arithmetic and logical expressions before; such expressions take a number of parameters as input, and returns a value:

```
int x = 7;
int y = 12;

int resultOfArithmeticExp = x * y;
bool resultOfLogicExp = x < y;
```

We can write these two expressions in a more formal way:

A) (int x, int y) => int
B) (int x, int y) => bool

This should be read as:

- Expression A takes two **int** parameters **x**, **y** as input, and returns an **int** value
- Expression B takes two **int** parameters **x**, **y** as input, and returns a **bool** value

So far, so good. Let us now write two slightly more complex expressions:

A) (int x, int y) => { return x * y; }
B) (int x, int y) => { return x < y; }

This should be read as:

- Expression A takes two **int** parameters **x**, **y** as input, and returns a function that calculates (x * y) and returns the result
- Expression B takes two **int** parameters **x**, **y** as input, and returns a function that calculates (x < y) and returns the result

The mind-bending part about this definition, is to realize that it is not describing invocation of the code in the expressions; it is describing a parameterized function, which we can "invoke" later on with specific arguments. These are examples of **lambda expressions**. It is probably still hard to see how this helps us, with regards to the previous problem of matching a **Car** object to specific values. Let's try to formulate a lambda expression closer to what we need:

(Car c) => { return (*code for a specific matching condition*); }

This is also a lambda expression. When will we write such an expression? Typically just when we need it. Let's see it in the context of an actual **Car** collection:

```
List<Car> carsList = new List<Car>();

// ...some Car objects are added to the list

string licensePlate = "CJ 32 802";
int price = 45000;

Predicate<Car> carMatchFunc = (Car c) => { return c.Price == price; };
Car matchCar = carsList.Find(carMatchFunc);

carMatchFunc = (Car c) => { return c.LicensePlate == licensePlate; };
matchCar = carsList.Find(carMatchFunc);
```

There are several things to take note of in this code. In the highlighted line, we have specified a lambda expression (the yellow part), which follows the syntax we introduced above. A particularly interesting feature is that the function part (after the => symbol) uses the local variable **price**. This has the very important consequence that the function only needs a **Car** object as parameter. So, this particular lambda expression takes a **Car** object as input, and returns a function that compares the **Price** property on the **Car** object with the value of the local variable **price**.

What about the blue part? The lambda expression in the yellow part returns a function taking a **Car** as input, and returning a **bool**. This kind of function is considered a type, just as any other type in C#. The type **Predicate<T>** – which is part of the .NET class library – is defined as being a function taking a parameter of type **T**, and returning a **bool** value. The variable **carMatchFunc** thus has the type *"a function taking a **Car** object as input, and returning a **bool** value"*. This is exactly what our lambda expression does, so the code is indeed valid.

In the next line, we make the call **carsList.Find**(**carMatchFunc**). If you study the documentation for the **List<T>** class, you will see that the **Find** method precisely takes a parameter of type **Predicate<T>**. The method will then return the (first) object for which the function returns **true**. The call **carsList.Find**(**carMatchFunc**) will thus return the first **Car** object for which the **Price** property equals the value of **price**. This is exactly the functionality we wanted! The last two lines illustrate that **carMatchFunc** is indeed just a variable, to which we can assign different values. In this case, we assign a different lambda expression to the variable (now matching on license plate), and call **Find** again.

Wrapping your mind around this functions-as-parameters idea is mentally challenging, and it requires you to think about functionality in a quite abstract way. Still, it is just another incarnation of principles we have seen before. Let's try to compare the code from above to some simpler code:

```
Predicate<Car> carMatchFunc = (Car c) => { return c.Price == price; };
Car matchCar = carsList.Find(carMatchFunc);

// ...is just as
int index = 12;
Car someCar = carsList[index];


carMatchFunc = (Car c) => { return c.LicensePlate == licensePlate; };
matchCar = carsList.Find(carMatchFunc);

// ...is just as
index = 16;
someCar = carsList[index];
```

In the first half of the code, we declare some local variables; two of type **Car**, one of type **int** named **index,** and one of type **Predicate<Car>** named **carMatchFunc**. We assign a specific value 12 to **index**, and a specific "value" (being a lambda expression) to **carMatchFunc**. We then use these variables – with the values currently assigned to them – to look up some **Car** objects. In the second half of the code, we assign new values to **index** and **carMatchFunc** – the value 16 and a new lambda expression, respectively – and once again use them to look up **Car** objects. So, we declare variables and assign values in the code shown above; those variables – and thus their current values – are then used inside the **List** methods.

For completeness, it should be mentioned that the **List** class contains several variants of the **Find** method. **Find** finds the first object for which the given predicate returns **true**, while the **FindLast** method finds the last object. We can easily imagine that the predicate will return **true** for more than one object, so a **FindAll** method is also available, which returns all objects for which the predicate returns **true**.

The syntax for lambda expressions described above can be considered the "fully dressed" version of the syntax. The compiler can often deduce the type of the parameters from the context, and you can also omit parentheses if the expression only takes a single parameter. A more succinct version of the code in our example is thus:

```
Car matchCar = carsList.Find(c => c.Price == price);
matchCar = carsList.Find(c => c.LicensePlate == licensePlate);
```

Delegates

The ability to parameterize functions with other functions is a powerful tool, when it comes to writing functions which are as general as possible. The **FindAll** method is a nice illustration of this idea; we can write a general method that performs an iteration through a collection of objects, picking out objects that match a certain condition, while making it possible for the caller to specify the exact condition. We also saw that the "signature" of a method (input parameters and return value) can be considered a type – the **Predicate<Car>** was an example of this – and we can therefore declare variables of such a type, and let them refer to e.g. a lambda expression. This was just a single example of a so-called **delegate**.

A **delegate** is essentially just a variable that can refer to a function. We claimed in the previous section that using a variable of such a "function-reference" type was quite similar to using e.g. an **int** variable, and that is almost true. The "almost" is added due to the fact that a delegate can in fact refer to a collection of functions! That is, a delegate can refer to zero, one or many functions. When a delegate is invoked, it will in turn invoke all of the functions to which it refers; it "delegates" the actual work to these functions.

The original syntax for creating and using delegates is a bit peculiar, so we will just show it briefly for completeness, and then proceed quickly to a more modern style. The **Predicate<T>** is an example of a this modern style.

Creating and using a delegate formally involves first creating a delegate type, and then a declaring a variable of that type:

```
delegate bool CarCheckDelegate(Car c);
private CarCheckDelegate theCarCheckDelegate = null;
```

The first line declares the type **CarCheckDelegate**, which returns a **bool** value, and takes a **Car** object as parameter. The next line then declares a variable **theCarCheckDelegate** of type **CarCheckDelegate,** and sets its initial value to **null**. With this in place, we can then assign a function reference to the variable:

```
theCarCheckDelegate = c => c.LicensePlate == licensePlate;
```

We have still not executed any code; to do this, we must "invoke" the delegate:

```
Car aCar = new Car("CJ 32 802", 5, 50000);
bool result = theCarCheckDelegate(aCar);
```

The last line will invoke all of the functions to which the delegate currently refers (in this example just one function).

The above code is valid and will work fine, but the somewhat lengthy syntax can be avoided by using some of the built-in, type-parameterized delegate types, like e.g. **Predicate<T>**. A handful of these delegate types exist:

Action **Action<T1>** **Action<T1, T2>** … **Action<T1,…,T16>**	An **Action** delegate has no (i.e. **void**) return type. All type parameters are thus the types of the input parameters. You can specify up to 16 input parameter types.
Func<TRes> **Func<T1, TRes>** **Func<T1, T2, TRes>** … **Func<T1,…,T16, TRes>**	A **Func** delegate has return type **TRes**. All type parameters except the last one are thus the types of the input parameters. You can specify up to 16 input parameter types.
Predicate<T>	A **Predicate** delegate always returns a **bool**, and takes one input parameter of type **T**.
Converter<TIn, TOut>	A **Converter** delegate always returns a value of type **TOut**, and takes one input parameter of type **TIn**.
Comparison<T>	A **Comparison** delegate takes two input parameters of type T, and should return an int value, following the same rules as specified for the **IComparer** interface.

It is fairly easy to see that the last three types of delegates are just special cases of the **Func** delegate. So why do they exist at all? Mostly for historic reasons... Knowing about **Action** and **Func** is usually sufficient to work with delegates, and it is generally recommended to use **Action** and **Func** instead of the older, more specific types.

We have already seen that use of the built-in delegate types makes the syntax a bit shorter. Declaring a delegate is typically a one-line operation now:

```
Func<Car, bool> theCarCheckDelegate = null;
```

Assignment to – and invocation of – the delegate follows the same syntax as before.

We claimed above that a delegate can refer to a collection of functions. The syntax for this is fairly straightforward. When the delegate has been declared as above, it refers to zero functions. Adding a reference to a function is done by using the **+=** operator:

```
theCarCheckDelegate += c => c.LicensePlate == licensePlate;
```

In general, you should use **+=** when adding a function reference to a delegate, since using **=** will remove the existing references! You can subsequently add more function references to the delegate, and even remove them again using the **-=** operator. When the delegate is invoked at some point, all functions to which the delegate refers are invoked, using the same arguments as specified in the delegate invocation.

Events

The idea of having variables referring to methods, and to invoke several methods “indirectly” through a delegate, does seem to be in contrast with the usual way of structuring code, where method calls are written explicitly. We have already seen that the delegate concept is a useful tool for turning code into a parameter, but that does not imply that delegates in general are a superior way to structure code. For applications where many “clients” (a “client” is here defined as a specific part of the application code) are interested in being informed about changes in other parts of the code, delegates can be suitable solution. The C# language construction called **events** are specifically designed for such scenarios. Events and delegates are closely related.

The only feature that distinguishes an event from an ordinary delegate is the use of the keyword **event** used in the declaration:

```
event Action<double> TemperatureChanged;
```

This declares an event of type **Action<double>** (no return value, takes one parameter of type **double**), in a way that looks very similar to how you declare delegates.

How could this event be used in practice? Suppose we have defined a class **TemperatureMonitor**, which e.g. monitors a temperature measurement device. The event declared above could then be part of this class. It would be declared either as a public instance field, or hidden behind methods/properties. In the code below, the first solution has been chosen:

```
public class TemperatureMonitor
{
	private double _temperature;

	public event Action<double> TemperatureChanged;

	public TemperatureMonitor()
	{
		TemperatureChanged = null;
	}

	// We assume somebody calls MonitorDevice at regular intervals.
	private void MonitorDevice()
	{
		// We assume GetTemperatureFromDevice retrieves
		// an actual temperature from some physical device.
		double newTemperature = GetTemperatureFromDevice();

		if (Math.Abs(newTemperature - _temperature) > 0.1)
		{
			_temperature = newTemperature;
			OnTemperatureChanged();
		}
	}

	private void OnTemperatureChanged()
	{
		TemperatureChanged?.Invoke(_temperature);
	}
}
```

The single line in the **OnTemperatureChanged** method is a standard code "idiom" (a short piece of code used very often), which reads "if **TemperatureChanged** is not **null**, call **Invoke** on **TemperatureChanged**, otherwise do nothing". **Invoke** is a method which is available on all variables of an event type, and calling it will call all of the functions which are currently referred to by the event.

Given the code in **MonitorDevice**, the net effect is thus that whenever the temperature changes (we have included a threshold of 0.1 degree, to avoid calling **OnTemperatureChanged** on very small changes), the method **OnTemperatureChanged** is called, which in turn invokes the event **TemperatureChanged**. When an event is invoked, is it often called to raise the event.

The idea is now than any client interested in knowing about temperature changes can "attach" a function to the event. The only requirement for the function is that it must match the type of the event, in this case **Action<double>**. We can imagine that e.g. a class responsible for displaying the temperature in a GUI would like to be notified about temperature changes:

```
public class GUIClient
{
      // Rest of class omitted for brevity

      public void TemperatureHasChanged(double temperature)
      {
            Console.WriteLine("Current temperature : " + temperature);
      }
}
```

The **TemperatureHasChanged** method can now be attached to the event:

```
      TemperatureMonitor monitor = new TemperatureMonitor();

      GUIClient client = new GUIClient();
      monitor.TemperatureChanged += client.TemperatureHasChanged;
```

Note that the attachment is done for objects; if we for some reason need several **GUIClient** objects, we must attach the **TemperatureHasChanged** method to the event for each of those objects.

It is not obvious why we need the **event** keyword at all, since an event seems to be just like any other delegate. That is true, except for a subtle difference, relating to a remark made earlier in this section. Note that in the highlighted line of code, we use the **+=** operator to attach a method to the event. This is the correct way to do this, since using **=** would remove all previously attached methods. However, for an event, we cannot use **=** at all! If we try to change **+=** to **=** in the above code, Visual Studio reports an error: *"The event **TemperatureChanged** can only appear on the left-hand side of **+=** and **-=**, except when used within the class **TemperatureMonitor**"*. Removing the **event** keyword from the declaration of **TemperatureChanged** in **TemperatureMonitor** – thereby turning it into an ordinary delegate – will "fix" the error, i.e. make the code compilable again. Adding the **event** keyword is thus a sort of fail-safe mechanism, preventing improper use of the event.

The concept of "event-driven applications" is not easy to grasp initially, since it is very different from the classic, sequential execution model. However, whenever you have an application where one part of the application needs to know immediately if something specific happens in another part, using events will often be an appropriate solution. The typical example is a GUI-rich application, where the GUI needs to invoke an action when a user performs a GUI operation. Events can also be used if changes in data (as in the example) need to be relayed to other parts of the code immediately.

Exercises

Exercise	**OOP.2.1**
Project	EmployeesV10
Purpose	See inheritance in action. Reorganize existing code to use inheritance. Call base class constructors.
Description	The project contains two existing classes **Teacher** and **ITSupporter**. They have quite a lot in common, so there is a lot of code duplication to get rid of.
Steps	Reorganize the code using inheritance 1. Create a new class **Employee**, that contains the common parts from **Teacher** and **ITSupporter**. 2. Let **Teacher** and **ITSupporter** inherit from the **Employee** class. The code in **Program.cs** should work as before. Remember that the derived classes will need to call the base class constructor.

Exercise	**OOP.2.2**
Project	WeaponShopV10
Purpose	Complete the implementation of two derived classes.
Description	The project contains the class **Weapon**, which is used as a base class for two specific weapon classes: **Wand** and **Axe**. A **Wand**: • Has a description, plus a minimum and maximum amount of damage dealt (see class definition). • Can be "enchanted". When enchanted, the wand deals double damage. A wand is initially <u>not</u> enchanted. An **Axe**: • Has a description, plus a minimum and maximum amount of damage dealt (see class definition). • Gets duller when used. This means that after each use, the minimum and maximum amount of damage dealt drops by three damage points. • Can be sharpened. This restores the minimum and maximum amount of damage dealt to the initial values.
Steps	1. In the **Wand** class, implement a property **IsEnchanted** of type **bool**. This property should represent whether or not the wand is currently enchanted. The property should have both a **get**-part and a **set**-part. 2. Also in the **Wand** class, implement a method **DamageFromWand**, which takes no parameters, and returns the amount of damage dealt by the wand (Hint: Use the method **CalculateDamage** from the base class). Remember the requirement about damage when enchanted. 3. In the **Axe** class, implement a method **DamageFromAxe**, which takes no parameters, and returns the amount of damage dealt by the axe (Hint: Use the method **CalculateDamage** from the base class). The method should also lower the values of maximum and minimum damage by three points. 4. Also in the **Axe** class, implement a method **Sharpen**, which takes no parameters, and returns no value. The method should reset the values of maximum and minimum damage to their initial values (Hint: use the constants defined in the class). 5. Open the **WeaponTester** class, uncomment all of the code currently commented out, and run the test. Do the results seem reasonable? 6. In **WeaponTester,** take a look at the methods **UseWand** and **UseAxe**. How are they similar? How are they different?

Exercise	**OOP.2.3**
Project	WeaponShopV20
Purpose	Modify the implementation of two derived classes, to take advantage of virtual/override construction. Work with polymorphic behavior.
Description	This exercise starts where the previous exercise left off. The base class **Weapon** and the derived classes **Axe** and **Wand** are functional; the goal in this exercise is to improve their structure.
Steps	1. In the **Weapon** class, change the method **protected int CalculateDamage()** to **public virtual int DealDamage()**. The body of the method should remain the same. 2. In the **Axe** class, change the method **public int DamageFromAxe()** to **public override int DealDamage()**. The body of the method must be changed at bit (Hint: call **base.DealDamage** instead of **CalculateDamage**). 3. In the **Wand** class, change the method **public int DamageFromWand()** to **public override int DealDamage()**. The body of the method must be changed at bit (Hint: call **base.DealDamage** instead of **CalculateDamage**). 4. Before proceeding, think about what you have done in steps 1) to 3). Why is it a good idea to define a method called **DealDamage** in the base class AND both of the derived classes? 5. In the **WeaponTester** class, replace the two methods **UseWand** and **Use-Axe** with a single method **UseWeapon**. The first parameter should now be of type **Weapon**. The rest of the method should work in a way similar to how **UseWand** and **UseAxe** works. 6. Still in the **WeaponTester** class, replace all usages of **UseWand** and **Use-Axe** with usage of the new method **UseWeapon** (you can delete the two original methods, to be sure you are not using them anymore). Run the test, and see that it works as before. 7. In the new method **UseWeapon**, we call **DealDamage** on a variable of type **Weapon**. Still, it seems like the versions of **DealDamage** defined in the derived classes get called (as they should). Why does this happen? What do we call this type of behavior?

Exercise	**OOP.2.4**
Project	RolePlayV23
Purpose	Override existing methods in derived class
Description	The project contains a working role-play system. Any character in the game is represented by an object of the class **Character**.
Steps	1. Get an overview of the application. The central class is the **Character** class, which implements a generic game character. Also note the code in **Program.cs**, where two teams with two members are set up for battle. 2. In the **Character** class, open the region *Virtual Properties and Methods*. Make sure you understand the purpose of these properties and methods. Note that you must <u>not</u> change anything in the **Character** class when you solve the next steps. 3. Create a class **Defender**, which derives from **Character**. A **Defender** has a 45 % chance of having the received damage reduced by 50 %. Implement this in the **Defender** class by <u>overriding</u> relevant properties and methods (i.e. the virtual properties/methods which are inherited from **Character**). Once you have created the class, update the code in **Program.cs** to include a **Defender** on each team. 4. Create a class **Damager**, which derives from **Character**. A **Damager** has a 40 % chance of dealing double damage. Implement this in the **Damager** class by <u>overriding</u> relevant properties and methods (i.e. the virtual properties/ methods which are inherited from **Character**). Once you have created the class, update the code in **Program.cs** to include a **Damager** on each team. 5. Reflect a bit on how we implemented the two derived classes. If we were allowed to make changes to the **Character** class, could we then implement the requirements from step 3) and 4) without having to create two new derived classes?

Exercise	**OOP.2.5**
Project	SimpleGeometry
Purpose	Override abstract methods. See polymorphic behavior in action.
Description	The project contains the (abstract) base class **Shape**, with an abstract property **Area**. The class also contains a static method **FindTotalArea**, that should calculate the total area of a list of shapes. The project also contains two derived classes **Circle** and **Rectangle**, which are not yet complete.
Steps	1. Implement the property **Area** correctly in **Circle** and **Rectangle**, using the available instance fields (if you need the value of π (pi), you can get it by writing **Math.PI**). 2. In the **Shape** class, implement the **FindTotalArea** method correctly, such that it finds the total area of a list of shapes. 3. Take a look at the code in **Program.cs**. It contains a small test of the classes in the project. Make sure you understand what the test does. 4. Run the application, and see if the output matches your expectations. 5. Why is it possible for the **FindTotalArea** method in the **Shape** class to work correctly, even though the **Area** property is defined as abstract?

Exercise	**OOP.2.6**
Project	FilteringV10
Purpose	Use interfaces to generalize code
Description	The project contains a class **Filter**, with a **FilterValues** method. The method filters out values higher than 10 from a list of integers. The project also contains an interface ***IFilterCondition***.
Steps	1. Figure out how to use the interface ***IFilterCondition*** to change the **Filter-Values** method, into a method that can filter a list of integers according to any condition. That is, the condition itself has to somehow become a parameter to the method. Try out your solution with a couple of conditions. 2. Figure out how you can apply several filter conditions to a list in a single method call. 3. Filtering is a very generic operation. Maybe some of the .NET collection classes already support filtering…?

Exercise	**OOP.2.7**
Project	CarDealershipV05
Purpose	Override methods from **Object** class.
Description	The project contains a simple class **Car**, which contains a few proper-ties. In **Program.cs**, we attempt to print out some **Car** objects, and perform some comparisons between **Car** objects.
Steps	1. Run the program as-is, and observe the result. Can you figure out when the comparisons return **true**? 2. In the **Car** class, uncomment the **Equals** method (only that method), and run the program again. What has changed? 3. Uncomment the rest of the code in the **Car** class, and run the program again. What has changed? 4. The printing of **Car** objects is still not very satisfying. In the **Car** class, over-ride the **ToString** method, so that it returns a string giving a reasonable description of the **Car** object (note that the **ToString** method should <u>not</u> call **Console.WriteLine**; it should only return a string to the caller). Run the program again, and see what difference it makes.

Exercise	**OOP.2.8**
Project	BankWithExceptions
Purpose	Add exceptions and exception handling to a project
Description	The project contains a class **BankAccount**. It defines a fairly straightforward bank account, but there are a few restrictions in it (see the code). One restriction is that the balance must not become negative. The project also contains three additional classes: • **IllegalInterestRateException** • **NegativeAmountException** • **WithdrawAmountTooLargeException** They are all exception classes, i.e. they inherit from **Exception**. The specific purpose of each exception class is described in the code. The **BankAccount** class already uses the **WithdrawAmountTooLargeException** class, to prevent that the balance becomes negative (see the **Withdraw** method)
Steps	1. Modify the code in the **BankAccount** class, such that the additional exception classes are used properly. 2. Study the code in **Program.cs**. Make sure you understand why the **try-catch** statements are included in the code. 3. Run the application, and test that the exceptions are now thrown and handled properly.

Exercise	**OOP.2.9**
Project	GenericRepository
Purpose	Create and use a Repository class based on Generics
Description	The project contains three simple domain classes **Car**, **Employee** and **Computer**, and two repository classes **CarRepository** and **Employee-Repository**.
Steps	1. Examine the implementation of **CarRepository** and **EmployeeRepository**. Take note of the similarities and differences between the classes. 2. There is currently no repository class for **Computer**. The next step is therefore to create a repository class for storing **Computer** objects. You can choose between two paths: a. ***The Path of Darkness***: Create a class named **ComputerRepository**, copy/paste code from one of the existing repository classes into the new class, and modify it to be able to handle **Computer** objects. Add code to **Program.cs** to test your new class. b. ***The Path of Light***: Create a type-parameterized class named **Repository**, which can be used for <u>any</u> domain class. Rewrite the code in **Program.cs** to use the new class for all three domain classes. 3. We now also want to be able to print out the content of (i.e. the objects stored in) a repository. Add this functionality to your repository class(es) by adding a method named **PrintAll**, and use it to print out the content of all three repositories. 4. We now also want to be able to retrieve the number of objects stored in a repository. Add this functionality to your repository class(es) by adding a property named **Count**. 5. Add a new domain class **Phone**, and repeat steps 2, 3 and 4 again. If you chose the Path of Darkness, feel free to reconsider your allegiance…

Exercise	**OOP.2.10**
Project	GenericsDogsAndCircles
Purpose	Increase cohesion and decrease coupling in the given project, by adding a type-parameterized class
Description	The project contains two unrelated domain classes **Dog** and **Circle**. The project also contains the class **ObjectComparer**, which contains methods for finding the “largest” **Dog** and **Circle** object out of three given objects.
Steps	1. Examine the three given classes, with particular focus on the **ObjectCom-parer** class. What are the problems with this class? 2. Let **Dog** inherit from **IComparable<Dog>** and implement the **CompareTo** method, as described in the notes. Compare according to **Weight**. 3. Let **Circle** inherit from **IComparable<Circle>** and implement the **Compare-To** method, as described in the notes. Compare according to **Area**. 4. Add a new class **BetterObjectComparer** to the project. The class should take one type parameter **T**, and have the constraint **where T : IComparable<T>** 5. Implement a method **Largest**, that takes three parameters of type **T**, and returns a reference to the “largest” object (hint: remember that you can now call **CompareTo** on a domain object, with another domain object as argument). 6. Rewrite the test code in **Program.cs** to use the new **BetterObjectCompa-rer** class. Test that your new code works as expected. 7. Why does this approach decrease coupling? Is there any coupling left between **BetterObjectComparer** and the domain classes?

Exercise	**OOP.2.11**
Project	GenericsDogsAndCircles (same project as used in previous exercise)
Purpose	Achieve further decoupling by using the **IComparer<T>** interface.
Description	The project starts out just in the previous exercise, with two domain classes **Dog** and **Circle**, and the **ObjectComparer** class.
Steps	1. Add a class **DogCompareByHeight**, which inherits from **IComparer<Dog>**. Implement the **Compare** method as outlined in the notes, and compare dogs by **Height**. 2. Add a class **CircleCompareByX**, which inherits from **IComparer<Circle>**. Implement the **Compare** method as outlined in the notes, and compare circles by **X** (x-coordinate). 3. Add a class **EvenBetterObjectComparer**. Note that the class does not need any type parameters. 4. In the **EvenBetterObjectComparer** class, implement a method **Largest<T>** (i.e. a method which takes a type parameter), which takes three parameters of type **T** and one parameter of type **IComparer<T>**. The method should return a reference to the "largest" object (hint: use the **Compare** method, which is available on the parameter of type **IComparer<T>**). 5. Rewrite the test code in **Program.cs** to use the new **EvenBetterObject-Comparer** class. Test that your new code works as expected. 6. What are the advantages of this solution, compared to the **BetterObject-Comparer** used in the previous exercise (Hint: Do the **Dog** and **Circle** classes need to implement any interfaces now)?

Exercise	**OOP.2.12**
Project	GenericsVariance
Purpose	Illustrate practical benefits from declaring type parameters as co-variant or contra-variant
Description	The project contains a simple class system for animals: An **Animal** base class, and two derived classes **Bird** and **Cat**. Furthermore, the project contains interfaces and classes for collections and collection processing.
Steps	1. Examine the two interfaces **ICollectionGet<T>** and **ICollectionSet<T>**. Pay particular attention to how the type parameter **T** is used in each interface. 2. Examine the class **Collection<T>**. It is a very simple collection class, that implements the two interfaces mentioned above. 3. Examine the **AnimalProcessor** class, which contains four methods. Pay particular attention to the type of the parameter to each method, and to the operations performed inside the methods. 4. Now open **Program.cs**, and examine the code. Notice the commented-out code, which contains 8 method calls (Case A to H). Before un-commenting the code, see if you can work out which method calls are valid, and which are not (Hint: Pay close attention to the specific type of the parameter in each call). 5. Un-comment the code. How many cases did you get right? 6. Now open the **ICollectionGet<T>** interface. Declare the type parameter **T** to be co-variant, by adding the keyword **out** just before the **T**, like this: **ICollectionGet<out T>**. 7. Go back to **Program.cs**. Which case(s) that were previously invalid are now valid? See if you understand why… 8. Now open the **ICollectionSet<T>** interface. Declare the type parameter **T** to be contra-variant, by adding the keyword **in** just before the **T**, like this: **ICollectionSet<in T>**. 9. Go back to **Program.cs**. Which case(s) that were previously invalid are now valid? See if you understand why… 10. Two cases remain invalid. Do you think we in any way could fix this by further adjustments of the interfaces? 11. Since the **Collection** class implements both **ICollectionGet<T>** and **ICollectionSet<T>**, wouldn't it be easier just to have a single interface **ICollection<T>**, containing all methods from the two interfaces? What would the consequences be?

Exercise	**OOP.2.13**
Project	LambdaAnimals
Purpose	Write a couple of small lambda expressions, and use them to filter out certain items in a collection
Description	The project contains a single class **Dog**. In **Program.cs**, a number of **Dog** objects are created and inserted into a **List<Dog>**. A method **ConditionalPrint** is also defined, which takes a list and a filtering condition as arguments.
Steps	1. At the indicated places in the code, call **ConditionalPrint** with a suitable lambda expression as argument, i.e. an expression which evaluates to true according to each of the three descriptions in the comments. 2. Create a new method **ConditionalPrint2**, which takes two lambda expressions as arguments. It should only print out those items that match both lambda expressions. 3. Imagine we have a list of conditions (in the form of lambda expressions) that we wish to use for filtering. See if you can create a **MultiConditionalPrint** method for that purpose.

Exercise	**OOP.2.14**
Project	ClockV20
Purpose	Use events to connect two domain objects.
Description	The project contains two classes: • **PulseGenerator**: This class can generate an event at regular time intervals, and allow other objects to be notified of these events. • **Clock**: This class simulates a simple 24-hour clock
Steps	1. Study the **PulseGenerator** class. Note in particular the **Pulse** event. What methods (with regards to parameters and return values) can be attached to this event? 2. Study the **Clock** class. Note in particular the methods **Tick** and **PrintTime**. How do you think they will be related to the **Pulse** event? 3. In **Program.cs**, a **PulseGenerator** object is created. Further down, the method call **theGenerator.Start(200)** is invoked. Between those two lines, create a **Clock** object, e.g. with a Danish text. 4. Just after creating the **Clock** object, attach the relevant methods from the object to the **Pulse** event on the **PulseGenerator** object. 5. Start the application, and see if the clock progresses as expected. 6. Create some additional **Clock** objects with different texts – and perhaps different tick factors – and attach them to the **Pulse** event. 7. Start the application again, and see if the additional clock objects behave as expected.

Exercise	OOP.2.15
Project	StockTrade
Purpose	Use events to connect various domain objects.
Description	The project contains classes for (very simplified) simulation of stock trading. Some objects will generate events, while other objects will need to be notified of these events.
Steps	1. Study the **TradeLog** class – it is quite simple ☺. 2. Study the **PulseGenerator** class. This class can generate an event at regular time intervals, and allow other objects to be notified of these events. 3. Study the **Stock** class. It simulates a real stock, including price changes. Notice the method **GenerateNewPrice**. Who do you think should call this method? Also notice the event **PriceChanged**. What is it used for? 4. Study the class **StockTrader**. It simulates a real stock trader, in a very simplified way. Note the method **DoTrade**. How is this method related to the **PriceChanged** event in the **Stock** class? We now want to connect the pieces to create a stock trade simulation. This is done in **Program.cs**. 5. After the creation of the **PulseGenerator** object – but before the call of **thePulseGenerator.Start** – create a few **Stock** objects. Choose some reasonable upper and lower limits for stock prices. 6. Create some **StockTrader** objects. Each stock should be traded by at least one stock trader. 7. Make sure that the **Stock** objects have their **GenerateNewPrice** method called, whenever the **PulseGenerator** object generates a **Pulse** event. 8. Make sure that the **StockTrader** objects are notified about changes in stock prices for the relevant stocks. 9. On each **Pulse** event, print out the current price of all stocks (hint: Create a lambda expression which prints out the stock prices, and attach it to the **Pulse** event). 10. On the **LastPulse** event, print out the entire trade log. 11. Run the application. Check to see if the trades obey the limits set for each stock trader. 12. See if you can extend the stock trader model, maybe by letting each stock trader trade more than one stock, have more advanced criteria for buying or selling, etc..