17. Accessing Data in Objects

This is the start of the lectures about data access and operations.

In this and the following sections we will discuss the various operations in classes, in particular how to access data which is encapsulated in objects. By data access we mean both reading (getting) and writing (setting).

In this material we use the word *operation* as a conceptual term. There is nothing called an "operations" in C#. Rather, there are methods, properties, indexers, operators etc. Thus, when we in this teaching material use the word operation it covers - in a broad sense - methods, properties, indexers, operators, events, delegates, and lambda expressions.

17.1. Indirect data access

Lecture 5 - slide 2

It is not a good idea to access the instance variables of a class directly from client classes. We have already discussed this issue in relatively great details in Section 11.3 and Section 11.5.

Data encapsulated in a class is not accessed **directly** from other classes

Rather, data is accessed **indirectly** via operations

So the issue is indirect data access instead of direct data access, when we work on a class from the outside (from other classes, the client classes). When we use indirect data access, the data are accessed through some procedure or function. This procedure or function serves as the *indirection* in between the client, which makes use of the data, and the actual data in the class. This "place of indirection" allows us to carry out checks and other actions in the slipstream of data access. In addition, the procedures and functions that serve as indirection, makes it possible to program certain compensations if the data representation is modified at a later point in time. With this, the client classes of a class C are more likely to survive future modifications of the data, which C encapsulates. It is possible to calculate the data instead of accessing it from variables in the memory.

The following summarizes why indirect data access is better than direct data access.

- Protects and shields the data
    - Possible to check certain conditions each time the data is accessed
    - Possible to carry out certain actions each time the data is accessed
- Makes it easier - in the future - to change the data representation
    - Via "compensations" programmed in the accessing operations
- Makes it possible to avoid the allocation of storage for some data
    - Calculating instead of storing

Protection and shielding may cause the data to be accessed in a conditional data structure. One particular shielding is provided by the precondition of an operation. If the condition does not hold, we may choose not to access the data. In this material, we discuss preconditions in the context of contracts in Chapter 50.

In some circumstances it may be convenient to carry out some action whenever the data of a class is accessed. This is probably most relevant when the data in the class is mutated (assigned to new values). Data values, for which we activate a procedure upon data access, are sometimes called *active values*.

Accessor compensation as a remedy of changing the representation of data is undoubtedly the most important issue. We illustrated this issue in the modifications of Program 11.2, as suggested in Exercise 3.3. In a nutshell, we can often fix the consequences of a shift in data representation by modifying the internals of the operations. By keeping the class interface unchanged all the direct and indirect clients of the affected class will survive. No changes are needed in the client classes. This is the effect of firewalls, as already discussed in Section 11.5.

17.2. Overview of data access in C#

Lecture 5 - slide 3

Below we summarize the various kinds of data access, as supported by operations in C#:

- Directly via public instance variables
    - **Never do that!**
- Indirectly via properties
    - Clients cannot tell the difference between access via properties and direct access of instance variables
- Indirectly via methods
    - Should be reserved for "calculations on objects"
- Indirectly via indexers
    - Provides access by means of the notation known from traditional array indexing
- Indirectly via overloaded operators
    - Provides access by means of the language-defined operator symbols

In the next chapter we will discuss properties in C#. Chapter 19 is about indexers. Methods will be discussed next in Chapter 20. Overloaded operators are treated in Chapter 21.

18. Properties

When a client of a class C accesses data in C via properties, the client of C may have the illusion that it accesses data directly. From a notational point of view, the client of C cannot tell the difference between access to a variable in C and access via a property.

Properties have not been invented in the process of creating C#. Properties have, in some forms, been used in Visual Basic and Delphi (which is a language in the Pascal family). Properties, in the sense discussed below, are not present in Java or C++. Java only allows data to be accessed directly or via methods. Therefore, in Java, it is always possible for clients of a class C to tell if data is accessed directly from a variable in C or indirectly via a method i C. In C#, it is not.

In this material we classify properties as operations, side by side with methods and similar abstractions. Underneath - in the Common Intermediate Language - properties are in fact treated as (getter and setter) methods.

18.1. Properties in C#

Lecture 5 - slide 7

When we use a property it looks like direct access of a variable. But it is not. A variable references to a stored location. A property activates a calculation which is encapsulated in an *abstraction*. The calculations that access data in a class C via properties should be efficient. If not, the clients of C are easily misled. Complicated, time consuming operations should be implemented in methods, see Chapter 20.

Let us first present a very simple, but at the same time a very typical example of properties. In Program 18.1 the `Balance` property accesses the private instance variable `balance`. Notice that the name of the property is capitalized, and that the name of the instance variable is not. This is a widespread convention in many coding styles.

```
using System;

public class BankAccount {

   private string owner;
   private decimal balance;

   public BankAccount(string owner, decimal balance) {
      this.owner = owner;
      this.balance = balance;
   }

   public decimal Balance {
     get {return balance;}
   }

   public void Deposit(decimal amount){
     balance += amount;
   }

   public void Withdraw(decimal amount){
     balance -= amount;
   }

   public override string ToString() {
      return owner + "'s account holds " +
            + balance + " kroner";
   }
}
```

Program 18.1 *A BankAccount class with a trivial Balance property together with Deposit and Withdraw methods.*

In class `BankAccount` it is natural to read the balance via a property, but is problematic to write the balance via a property. Therefore, there is no setter in the `Balance` property. Instead, `Deposit` and `Withdraw` operations (methods) are used. In general we should carefully consider the need for readability and writablity of individual instance variables.

The public `Balance` property as programmed in Program 18.1 provides for read-access to the private instance `balance` variable. You may complain that this is a complicated way of making the instance variable public. What is important, however, is that at a later point in the program evolution process we may change the private data representation. We may, for instance, eliminate the instance variable `balance` entirely, but keep the interface to clients - the `Balance` property - intact. This is illustrated in Program 18.2 below.

```
using System;

public class BankAccount {

   private string owner;
   private decimal[] contributions;
   private int nextContribution;

   public BankAccount(string owner, decimal balance) {
      this.owner = owner;
      contributions = new decimal[100];
      contributions[0] = balance;
      nextContribution = 1;
   }

   public decimal Balance {
     get {decimal result = 0;
          foreach(decimal ctr in contributions)
             result += ctr;
          return result;
         }
   }

   public void Deposit(Decimal amount){
     contributions[nextContribution] = amount;
     nextContribution++;
   }

   public void Withdraw(Decimal amount){
     contributions[nextContribution] = -amount;
     nextContribution++;
   }

   public override string ToString() {
      return owner + "'s account holds " +
            + Balance + " kroner";
   }
}
```

Program 18.2 *A BankAccount class with a Balance property - without a balance instance variable.*

The interesting thing to notice is that the balance of the bank account now is represented by the decimal array called `contributions` in line 6 of Program 18.2. The `Balance` property in line 16-22 accumulates the contributions in order to calculate the balance of the account.

From a client point of view we can still read the balance of the bank account via the `Balance` property. Underneath, however, the implementation of the `Balance` getter in line 16-22 of Program 18.2 has changed a lot compared to line 14 of Program 18.1. We show a simple client program in Program 18.3, and its output in Listing 18.4 (only on web).

The client program in Program 18.3 can both be used together with `BankAccount` in Program 18.1 and Program 18.2. Thus, the client program has no awareness of the different representation of the balance in the two versions of class `BankAccount`. The only thing that matters in the relation between class `BankAccount` and its client class is the client interface of class `BankAccount`.

```
using System;

class C{

  public static void Main(){
    BankAccount ba = new BankAccount("Peter", 1000);
    Console.WriteLine(ba);

    ba.Deposit(100);
    Console.WriteLine("Balance: {0}", ba.Balance);

    ba.Withdraw(300);
    Console.WriteLine("Balance: {0}", ba.Balance);

    ba.Deposit(100);
    Console.WriteLine("Balance: {0}", ba.Balance);
  }

}
```

Program 18.3 *A client program.*

Above we have discussed *getting of instance variables* in objects. In the `BankAccount` class we have seen how to access to the `balance` instance variable via a getter in the property `Balance`. *Technically*, it is also possible to change the value of the balance instance variable by a *setter*. *Conceptually*, we would rather prefer to update the bank accounts by use of the methods `Deposit` and `WithDraw`. Nevertheless, here is the `Balance` property with both a getter and a setter.

```
public decimal Balance {
  get {return balance;}
  set {balance = value;}
}
```

The setter is activated in an assignment like `b.Balance =` *expression*`;` The important thing to notice is that the property `Balance` is located at the left-hand side of the assignment operator. The value of *expression* is bound to the pseudo variable `value` in the property, and as it appears in the setter, the value of `value` is assigned to the instance variable `balance`.

Properties can also be used for getting and setting fields of struct values. In addition, properties can be used to get and set static variables in both classes and structs.

This ends the essential examples of the `Balance` property of class `BankAccount`. In the web version of the material we provide yet another variation of the `Balance` property in class `BankAccount`. In this example, we enforce a *strict alternation between getting and setting* the balance of a bank account. Please consult the web edition for details.

This ends our discussion of the exotic variation of class `BankAccount`.

Exercise 5.1. *A funny BankAccount*

In this exercises we provide a version of class `BankAccount` with a "funny version" of the `Balance` property. You should access the exercise via the web version, in order to get access to the source programs involved.

Study the `Balance` property of the funny version of class `BankAccount`.

Explain the behaviour of the given client of the funny BankAccount.

Next test-run the program and confirm your understanding of the two classes.

Please notice how difficult it is to follow the details when properties - like `Balance` in the given version of class `BankAccount` - do not pass data directly to and from the instance variables.

18.2. Properties: Class Point with polar coordinates

Lecture 5 - slide 8

We will now look at another very realistic example of properties in C#.

The example in Program 18.8 is a continuation of the `Point` examples in Program 11.2. Originally, and for illustrative purposes, in Program 11.2 we programmed a simple point with public access to its x and y coordinates. We never do that again. The x and y coordinates used in Program 18.8 are called *rectangular coordinates* because they delineate a rectangle between the point and (0,0).

In the class `Point` in Program 18.8 we have changed the data representation to *polar coordinates*. In the paper version of the material we show only selected parts of the class. (We have, in particular, eliminated a set of static methods that convert between rectangular and polar coordinates). In the web version the full class definition is included. Using polar coordinates, a point is represented as a radius and an angle. In addition, and as emphasized with **purple** in Program 18.8 we have programmed four properties, which access the polar and the rectangular coordinates of a point. The properties `Angle` and `Radius` are, of course, trivial, because they just access the underlying private instance variables. The properties `X` and `Y` require some coordinate transformations. We have programmed all the necessary coordinate transformations in static private methods of class `Point`. In the web edition of the material these methods are shown at the bottom of class `Point`.

```
// A versatile version of class Point with Rotation and internal methods
// for rectangular and polar coordinates.

using System;

```

```
public class Point {

  public enum PointRepresentation {Polar, Rectangular}

  private double r, a;            // Polar data representation

  public Point(double x, double y){
     r = RadiusGivenXy(x,y);
     a = AngleGivenXy(x,y);
  }

  public double X {
    get {return XGivenRadiusAngle(r,a);}
  }

  public double Y {
    get {return YGivenRadiusAngle(r,a);}
  }


  public double Radius {
    get {return r;}
  }

  public double Angle{
    get {return a;}
  }

  // Some constructors and methods are not shown

}
```

Program 18.8 *Class Point with polar data representation.*

Exercise 5.2. *Point setters*

In the `Point` class on the accompanying slide we have shown how to program getter properties in the class `Point`. Extend the four properties with setters as well. The new version of this class will support mutable points.

Write a small client program that demonstrates the use of the setter properties.

Hint: Please be aware that the revised class should allow us to get and set the rectangular and polar coordinates (x, y, angle, radius) of a point independent of each other. You should first consider what it means to do so.

18.3. Automatic Properties

Lecture 5 - slide 9

Many of the properties that we write are trivial in the sense that they just get or set a single instance variable. It is tedious to write such trivial properties. It may be possible to ask the programming environment to help with creation of trivial properties. Alternatively in C# 3.0, it is possible for the compiler to generate the trivial properties automatically from short descriptions.

Let us study an example, which extends the initial bank account example from Program 18.1. The example is shown in Program 18.9 and the translation done by the compiler is shown in Program 18.10.

```
1  using System;
2  
3  public class BankAccount{
4  
5    // automatic generation of private instance variables
6  
7    public BankAccount(string owner, decimal balance){
8      this.Owner = owner;
9      this.Balance = balance;
10   }
11 
12   public string Owner {get; set;}
13 
14   public decimal Balance {get; set;}
15 
16   public override string ToString(){
17     return  Owner  + "'s account holds " +  Balance  + " kroner";
18   }
19 }
```

Program 18.9   *Class BankAccount with two automatic properties.*

Based on the formulations in line 12 and 14, the compiler generates the "real properties" shown below in line 13-21 of Program 18.10.

As an additional and important observation, it is no longer necessary to define the private instance variables. The compiler generates the private "backing" instance variables automatically. In terms of the example, the lines 5-6 in Program 18.10 are generated automatically.

As a consequence of automatically generated instance variables, the instance variables cannot be accessed in the class. The names of the instance variables are unknown, and therefore they cannot be used at all! Instead, the programmer of the class accesses the hidden instance variables through the properties. As an example, the owner and balance are accessed via the properties `Owner` and `Balance` in line 17 of the `ToString` method in Program 18.9.

```
1  using System;
2  
3  public class BankAccount{
4  
5    private string _owner;
6    private decimal _balance;
7  
8    public BankAccount(string owner, decimal balance){
9      _owner = owner;
10     _balance = balance;
11   }
12 
13   public string Owner {
14     get {return _owner;}
15     set {_owner = value;}
16   }
17 
18   public decimal Balance {
19     get {return _balance;}
20     set {_balance = value;}
21   }
```

```

  public override string ToString(){
    return  Owner  + "'s account holds " +  Balance  + " kroner";
  }
}
```

Program 18.10 *An equivalent BankAccount class without automatic properties.*

Because properties, internally in a class, play the role of instance variables, it is hardly meaningful to have an automatic property with a getter and no setter. The reason is that the underlying instance variable cannot be initialized - simple because its name is not available. Similarly, an automatic property with a setter, but not getter, would not make sense, because we would not be able to access the underlying instance variable. Therefore the compiler enforces that an automatic property has both a getter and a setter. It is possible, however, to limit the visibility of either the getter or setter (but not both). As an example, the following definition of the `Balance` property

```
public string Balance {get; private set;}
```

provides for public reading, but only writing from within the class. In class `BankAccount`, this is probably what we want. It is OK to access the balance from outside, but we the account to be updated by use of either the `Deposit` method or the `Withdraw` method.

Finally, let us observe that the syntax of automatic properties is similar to the syntax used for abstract properties, see Section 30.3.

Automatic properties: Reflections and recommendations. FOCUS BOX 18.1

I recommend that you use automatic properties with some care! It is worth emphasizing that a class almost always encapsulates some state - instance variables - some of which can be accessed by properties. It feels strange that we can program in a way - with automatic properties - without ever declaring the instance variables explicitly. And it feels strange that we never, from within the class, refers to the instance variables.

Automatic properties are useful in the early life of a class, where we allows for direct (one-to-one) access to the instance variables via methods or properties. In an early version of the class `Point` of Section 11.6, this was the case. (We actually started with public instance variables, but this mistake is taken care of in Exercise 3.3).

Later on, we may internally invent a more sophisticated data representation - or we may change our mind with respect to the data representation. In the `Point` class we went from a rectangular representation to a polar representation. The data representation details should always be a private and internal concern of the class. When this change happens we have to introduce instance variables, exactly as described in Section 11.8. When doing so we have to get rid of the automatic properties. The introduction of instance variables and the substitution of the automatic properties with 'real properties' represent internal changes to the `Point` class. The clients of `Point` will not be affected by these changes. *This is the point!* In the rest of the lifetime of class `Point`, it is unlikely that automatic properties will be 'part of the story'.

Automatic properties contribute to a C# 3.0 *convenience layer* on top of already existing means of expressions.

18.4. Object Initialization via Properties

Lecture 5 - slide 10

In this section we will see that property setters in C#3.0 can be used for initialization purposes in the slipstream of constructors. The properties that we use for such purposes can be automatic properties, as discussed in Section 18.3.

Let us again look at a simple `BankAccount` class, see Program 18.11. The class has two automatic properties, backed by two instance variables, which we cannot access. In addition, the class has two constructors, cf. Section 12.4.

```
using System;

public class BankAccount{

  // automatic generation of private instance variables

  public BankAccount(): this("NN") {
  }

  public BankAccount(string owner){
    this.Balance = 0;
    this.Owner = owner;
  }

  public string Owner {get; set;}

  public decimal Balance {get; set;}

  public override string ToString(){
    return Owner + "'s account holds " + Balance + " kroner";
  }
}
```

Program 18.11 *Class BankAccount - with two constructors.*

Below, in Program 18.12 we make an instance of class `BankAccount` in line 6. As emphasized in **purple** the owner and balance is initialized by a so-called *object initializer*, in curly brackets, right after `new BankAccount`. The initializer refers the setter of the automatic properties `Owner` and `Balance`. All together we have made a new bank account by use of the parameterless constructor. The initialization is done by the setters of the `Owner` and the `Balance` properties.

In line 7 of Program 18.12 we make another instance of class `BankAccount`, where `Owner` is initialized via the actual parameter "Bill" passed to the constructor, and the balance is initialized via an object initializer in curly brackets.

```
1  using System;
2  
3  public class Client {
4  
5    public static void Main(){
6      BankAccount ba1 = new BankAccount{Owner = "James", Balance = 250},
7                  ba2 = new BankAccount("Bill"){Balance = 1200};
8  
9  
10     Console.WriteLine(ba1);
11     Console.WriteLine(ba2);
12   }
13 
14 }
```

Program 18.12 *A client of class BankAccount with an object initializer.*

The compiler translates line 6 of Program 18.12 to line 6-8 in Program 18.13. Similarly, line 7 of Program 18.12 is translated to line 10-11 in Program 18.13.

```
1  using System;
2  
3  public class Client {
4  
5    public static void Main(){
6      BankAccount ba1 = new BankAccount();
7      ba1.Owner = "James";
8      ba1.Balance = 250;
9  
10     BankAccount ba2 = new BankAccount("Bill");
11     ba2.Balance = 1200;
12 
13     Console.WriteLine(ba1);
14     Console.WriteLine(ba2);
15   }
16 
17 }
```

Program 18.13 *An equivalent client of class BankAccount without object initializers.*

Let us, finally, elaborate the example with the purpose of demonstrating *nested object initializers*. In Program 18.14 we show the classes `BankAccount` and `Person`. As can be seen, an instance of class `BankAccount` refers to an owner which is an instance of class `Person`. It turns out to be crucial for the example that the `Owner` of a `BankAccount` refers to a `Person` object (more specifically, that it is not a `null` reference).

```
using System;

public class BankAccount{

  public BankAccount(){
     this.Balance = 0;
     this.Owner = new Person();
  }

  public Person Owner {get; set;}
  public decimal Balance {get; set;}
12
13   public override string ToString(){
14     return Owner + "'s account holds " + Balance + " kroner";
15   }
16 }
17
18 public class Person {
19   public string FirstName {get; set;}
20   public string LastName {get; set;}
21
22   public override string ToString(){
23     return FirstName + " " + LastName;
24   }
25 }
```

Program 18.14   *Class BankAccount and class Person.*

In the `Client` class, see Program 18.15 we make two `BankAccounts` , `ba1` and `ba2`. The initializers, emphasized in line 7-9 and 12-14, initialize the `Owner` of a `BankAccount` with *nested object initializers*. Notice that there is no new operator in front of `{FirstName = ..., LastName = ...}`. The `Person` object already exists. In this example, it is instantiated in line 7 of Program 18.14 together with the `BankAccount`.

```
1  using System;
2
3  public class Client {
4
5    public static void Main(){
6      BankAccount ba1 = new BankAccount{
7                              Owner = {FirstName = "James",
8                                       LastName = "Madsen"},
9                              Balance = 250},
10
11                 ba2 = new BankAccount{
12                              Owner = {FirstName = "Bill",
13                                       LastName = "Jensen"},
14                              Balance = 500};
15
16     Console.WriteLine(ba1);
17     Console.WriteLine(ba2);
18   }
19
20 }
```

Program 18.15   *A client of class BankAccount with nested object initializers.*

The use of property names (setters) in object initializers gives the effect of *keyword parameters*. Keyword parameters refer to a formal parameter name in the actual parameter list. Keyword parameters can be given in any order, as a contrast to *positional parameters*, which must given in the order dictated by the formal parameter list. In addition, the caller of an abstraction that accepts keyword parameters can choose not to pass certain keyword parameters. In that case, defaults will apply. In case of C#, such default values can be defined within the body of the constructors.

18.5. Summary of properties in C#

Lecture 5 - slide 11

The syntax of property declarations is shown in Syntax 18.1. Both the get part and the set part are optional. Syntactically, the signature a property declaration is like a method declaration without formal parameters. As can be seen, the syntax of a property body is very different from the syntax of a method body.

```
 modifiers return-type property-name{
  get {body-of-get}
  set {body-of-set}
}
```

Syntax 18.1 *The syntax of a C# property. Here we show both the getter and setter. At least one of them must be present.*

The following summarizes the characteristics of properties in C#:

- Provides for either read-only, write-only, or read-write access
- Both instance and class (static) properties make sense
- Property setting appears on the left of an assignment, and in **++** and **--**
- Trivial properties can be defined "automatically"
- Properties should be fast and without unnecessary side-effects

The following observations about property naming reflect a given coding style. A coding style is not enforced by the compiler.

A C# property will often have the same name as a private data member

The name of the property is capitalized - the name of the data member is not

19. Indexers

From a notational point of view it is often attractive to access the data, encapsulated in a class or struct, via conventional array notation. Indexers are targeted to provide array notation on class instances and struct values.

Indexers can be understood as a specialized kind of properties, see Chapter 18. Both indexers and properties are classified as *operations* in this material, together with methods and other similar abstractions.

19.1. Indexers in C#

Lecture 5 - slide 13

Indexers allow access to data in an object with use of array notation

The important benefit of indexers is the notation they make available to their clients.

Let us assume that **`v`** is a variable that holds a reference to an object *obj*. With use of methods we can access data in *obj* with **`v.method(parameters)`**. In Chapter 18 we introduced properties and the property access notation **`v.property`**. We will now introduce the notation **`v[i]`**, where **`i`** typically (but not necessarily) is some integer (index) value.

We will start with an artificial ABC example in Program 19.1 which tells how to define an indexer in a class that encapsulates three instance variables `d`, `e`, and `f`. The indexer is used in a client class in Program 19.2. The example introduces the indexer abstraction, but it is not a typical use of an indexer.

As it can be seen in Program 19.1 an indexer must be named "this". Like a property, it has a getter and a setter part.

The getter is activated when we encounter an expression `a[i]`, where `a` is a variable of type `A`. The body of the getter determines the value of `a[i]`.

The setter is activated when we encounter an assignment like **`a[i]`** = *expression*. The value of *expression* is bound to the implicit parameter named `value`. The body of the setter determines the effect on the instance variables in *obj* upon execution of `a[i]` = *expression*.

In Program 19.1 `a[1]` accesses the instance variable `d`, `a[2]` accesses the instance variable `e`, and a[3] accesses the instance variable `f`. This is not a typical arrangement, however. Most often, indexers are used to access members of a data collection.

```
using System;

public class A {
  private double d, e, f;

  public A(double v){
    d = e = f = v;
  }

  public double this [int i]{
   get {
     switch (i){
       case 1: {return d;}
       case 2: {return e;}
       case 3: {return f;}
       default: throw new Exception("Error");
     }
   }
   set {
     switch (i){
       case 1: {d = value; break;}
       case 2: {e = value; break;}
       case 3: {f = value; break;}
       default: throw new Exception("Error");
    }
   }
  }

  public override string ToString(){
    return "A: " + d + ", " + e + ", " + f;
  }

}
```

Program 19.1 *A Class A with an indexer.*

Program 19.2 shows the indexer from Program 19.1 in action. First, in line 9, we illustrate the three setters, where `a[i]` occurs at the left-hand side of the assignment symbol. Following that, in line 11, we illustrate two getters. The output of Program 19.2 is shown in Listing 19.3 (only on web).

```
1  using System;
2  
3  class B {
4  
5    public static void Main(){
6      A a = new A(5);
7      double d;
8  
9      a[1] = 6; a[2] = 7.0; a[3] = 8.0;
10 
11     d = a[1] + a[2];
12 
13 
14     Console.WriteLine("a: {0}, d: {1}", a, d);
15   }
16 }
```

Program 19.2 *A client of A which uses the indexer of A.*

As an additional example of indexers we will study the class `BitArray`. This example is only present in the web-version of the material.

19.2. Associative Arrays

Lecture 5 - slide 14

In Section 19.1 we showed how to index an object with a single integer index. In this section we will demonstrate that the indexing value can have an arbitrary type. Thus, the type of `obj` in `a[obj]` can be an arbitrary type in C#, for instance a type we program ourselves.

An *associative array* is an array which allows indexing by means of arbitrary objects, not just integers

An associative arrays maps a set of objects (the indexing objects, keys) to another set of objects (the element objects).

In Program 19.6 we illustrate how to index instances of class A with strings instead of integers. If you understood Program 19.1 and Program 19.2 it will also be easy to understand Program 19.6 and Program 19.7. Notice in this context that C#, very conveniently, allows switch control structures to switch on strings. The program output is shown in Listing 19.8 (only on web).

```
using System;

public class A {
  private double d, e, f;

  public A(double v){
    d = e = f = v;
  }

  public double this [string str]{
   get {
     switch (str){
       case "d": {return d;}
       case "e": {return e;}
       case "f": {return f;}
       default: throw new Exception("Error");
     }
   }
   set {
     switch (str){
       case "d": {d = value; break;}
       case "e": {e = value; break;}
       case "f": {f = value; break;}
       default: throw new Exception("Error");
    }
   }
  }

  public override string ToString(){
    return "A: " + d + ", " + e + ", " + f;
  }

}
```

Program 19.6 *The class A indexed by a string.*

```
using System;

class B {

  public static void Main(){
    A a = new A(5);
    double d;

    a["d"] = 6; a["e"] = 7.0; a["f"] = 8.0;

    d = a["e"] + a["f"];    // corresponds to d = a.d + a.e
                            // in case d and e had been public

    Console.WriteLine("a: {0}, d: {1}", a, d);
  }
}
```

Program 19.7 *A client of A which uses the string indexing of A.*

We have seen that it makes sense to index with strings, and more generally with an arbitrary instance of a class. In fact, it is possible to base the indexing on two or more objects. This is, of course, important if we index multi-dimensional data structures.

Associative arrays are in C# implemented by means of hashtables in dictionaries

In the lecture about collections, see Chapter 46, we will see how to make use so-called dictionaries (typically implemented as hash tables) for efficient data structures that map a set of objects to another set of objects. Indexers, as discussed in this section chapter, provide a convenient surface notation to deal with such dictionaries. In Section 46.2 the indexer prescribed by the generic interface **`IDictionary<K,V>`** accesses objects of type `V` via an index of type `K`.

19.3. Summary of indexers in C#

Lecture 5 - slide 15

Here follows a syntax diagram of indexers:

```
 modifiers return-type this[formal-parameter-list]
  get {body-of-get}
  set {body-of-set}
}
```

Syntax 19.1 *The syntax of a C# indexer*

It is similar to the syntax diagram of properties, as shown in Syntax 18.1

The main characteristics of indexers are as follows:

- Provide for indexed read-only, write-only, or read-write access to data in objects
- Indexers can only be instance members - not static
- The indexing can be based on one, two or more formal parameters
- Indexers can be overloaded
- Indexers should be without unnecessary side-effects
- Indexers should be fast

20. Methods

Methods are the most important kind of operations in C#. Methods are more fundamental than properties and indexers. We would be able to do object-oriented programming without properties and indexers (by implementing all properties and indexers as methods), but not without methods. In Java, for instance, there are methods but no properties and no indexers.

A method is a procedure or a function, which is part of a class. Methods (are supposed to) access (operate on) the data, which are encapsulated by the class. Methods should be devoted to nontrivial operations. Trivial operations that just read or write individual instance variables should in C# be programmed as properties.

We have already in Section 11.9 and Section 11.11 studied the fundamentals of methods in an object-oriented programming language. In these sections we made the distinction between instance methods and class methods. Stated briefly, instance methods operate on instance variables (and perhaps class variables as well). Class methods (called static methods in C#) can only operate on class variables (static variables in C#).

When we do object-oriented programming we organize most data in instances of classes (objects) and in values of struct types. We only use class related data (in static variables) to a lesser degree. Therefore instance methods are more important to us than class methods. In the rest of this chapter we will therefore focus on instance methods.

This chapter is long because we have to cover a number of different parameter passing modes. If you only need a basic understanding of methods and the most frequently used parameter passing mode - call-by-value parameters - you should read until (and including) Section 20.4 and afterwards proceed to Chapter 21.

20.1. Local variables in methods

Lecture 5 - slide 18

Local variables of a method are declared in the *statements* part of the block relative to Syntax 20.1. Local variables are short lived; They only exists during the activation of the method.

```
modifiers return-type method-name(formal-parameter-list){
  statements
}
```

Syntax 20.1 *The syntax of a method in C#*

You should notice the difference between *local variables* and *parameters*, which we discuss below in Section 20.2. Parameters are passed and initialized when the method is activated. The initial value comes from an actual parameter. Local variables are introduced in the *statements* part (the body) of the method, and as explained below they may - or may not - be initialized explicitly.

You should also notice the difference between *local variables* and *instance variables* of a class. A local variable only exists in a single call of the method. An instance variable exists during the lifetime of an object.

Local variables

- May be declared anywhere in the block
    - Not necessarily in the initial part of the block
- Can be initialized by an initializer
- Can alternatively be declared without an initializer
    - No default value is assigned to a local variable
        - Different from instance and class variables which are given default values
    - The compiler will complain if a local variable is referred without prior assignment

In the program below we contrast instance variables of the class `InitDemo` with local variables in the method `Operation` of `InitDemo`. The **purple** instance variables are implicitly initialized to their default values. The **blue** local variables in `Operations` are not. The program does not compile. In line 18 and 19 the compiler will complain about use of unassigned local variables.

```
1  using System;
2  
3  class InitDemo{
4  
5    private int intInstanceVar;
6    private bool boolInstanceVar;
7  
8    public void Operation(){
9      int intLocalVar;
10     bool boolLocalVar;
11 
12     Console.WriteLine("intInstanceVar: {0}. boolInstanceVar: {1}",
13                       intInstanceVar,
14                       boolInstanceVar);
15 
16     // Compile time errors:
17     Console.WriteLine("intLocalVar: {0}. boolLocalVar: {1}",
18                       intLocalVar,
19                       boolLocalVar);
20 
21   }
22 
23   public static void Main(){
24     new InitDemo().Operation();
25   }
26 
27 }
```

Program 20.1 *Local variables and instance variables - initialization and default values.*

In C#, Java and similar languages there is no such thing as a *global variable*. This is often a problem for programmers who are used to pass data around via global variables. If you really, really need a global variable in C#, the best option is to use a class (static) variable in one of your top-level classes. In general, however, it is a better alternative to pass data around via parameters (such as parameters to constructors).

20.2. Parameters

Lecture 5 - slide 19

As a natural counterpart to Syntax 18.1 (of properties) and Syntax 19.1 (of indexers) we have shown the syntactical form of a method in Syntax 20.1. The syntactical characteristic of methods, in contrast to properties and indexers, is the formal parameters in ordinary, soft parentheses: `(...)`. Even a method with no parameters must have an empty pair of parentheses ( ) - in both the method definition and in the method activation. Properties have no formal parameters, and indexers have formal parameters in brackets: `[...]`.

We will now discuss parameter passing in general. The following introduces *formal parameters* and *actual parameters*.

Actual parameters occur in a call. *Formal parameters* occur in the method declaration. In general, actual parameters are expressions which are evaluated to *arguments*. Depending on the kind of parameters, the arguments are somehow associated with the formal parameters.

C# offers several different parameter passing modes:

- Value parameters
  - The default parameter passing mode, without use of a modifier
- Reference parameters
  - Specified with the **`ref`** modifier
- Output parameters
  - Specified with the **`out`** modifier
- Parameter arrays
  - Value parameters specified with the **`params`** modifier

Far the majority of the parameters in a C# program are passed by value. Thus, the use of value parameters is the most important parameter passing technique for you to understand. Value parameters are discussed in Section 20.3 - Section 20.5.

Reference and output parameters are closely related to each other, and they are only rarely used in object-oriented C# programs. Output parameter can be seen as a restricted version of reference parameters. Reference parameters stem from *variable parameters* (`var` parameters) in Pascal. Reference parameters are discussed in Section 20.6 and out parameters are discussed in Section 20.7.

Parameter arrays cover the idea that a number of actual value parameters (of the same type) are collected into an array. In this way, parameter arrays provide for a more sophisticated correspondence between the arguments and the formal parameters. C# parameter arrays are discussed in Section 20.9.

It is, in general, an issue how a given actual parameter (or argument) is related to a formal parameter. This is called *parameter correspondence*. With value, **`ref`** and **`out`** parameter we use *positional parameter correspondence*. This is simple. The first formal parameter is related to the first actual parameter, the second to the second, etc. With parameter arrays, a number of actual parameters - all remaining actual parameters - correspond to a single formal parameter.

In general, there are other parameter correspondences, most notable *keyword parameters*, where the name of the formal parameter is used together with the actual parameter. Keyword parameters are not used directly in C#. But as we have seen in Section 18.4 a kind of keyword parameters is used when properties are used for object initialization in the context of the `new` operator.

In Section 6.9 - Program 6.20 - we have shown a program example that illustrate multiple parameter passing modes in C#. If you wish the ultra short description of parameter passing in C# you can read Section 6.9 instead of Section 20.3 - Section 20.9.

20.3. Value Parameters

Lecture 5 - slide 20

Value parameters are used for input to methods

When the story should told briefly, *call-by-value parameters* (or just *value parameters*) are used for input to methods. The output from a method is handled via the value returned from the method - via use of **`return`**. This simple version of the story is true for the majority of the methods we write in our classes. But as we will see already in Section 20.4 it is also possible to handle some kinds of output via references to objects passed as value parameters.

Value parameter passing works in the following way:

- A formal parameter corresponds to a local variable
- A formal parameter is initialized by the corresponding argument (the value of the actual parameter expression)
  - A *copy* of the argument is bound to the formal parameter
- Implicit conversions may take place
- A formal parameter is assignable, but with no effect outside the method

We have already seen many examples of value parameters. In case you want to review typical examples, please consult the `Move` method of class `Point` in Program 11.2, the `Withdraw` and `Deposit` methods of class `BankAccount` in Program 11.8, and the `AgeAsOf` method in class `Person` in Program 16.10.

20.4. Passing references as value parameters

Lecture 5 - slide 21

Care must be taken if we pass references as value parameters

Most of the data we deal with in object-oriented C# programs are represented as instances of classes - as objects. This implies that such data are accessed by references. Again and again we pass such references as value parameters to methods. Therefore we must understand - in details - what happens.

Here is the short version of the story. Let us assume that send the message `DayDifference` to a `Date` object with another `Date` object as parameter:

```
someDate.DayDifference(otherDate)
```

`Date` is a class, and therefore both `someDate` and `otherDate` hold references to `Date` objects. It is possible for the `DayDifference` method to mutate the `Date` object referred by `otherDate`, despite the fact that the parameter of `DayDifference` is a value parameter. It is not, however, possible for `DayDifference` to modify value of the variable (actual parameter) `otherDate` as such.

In the web-version we present the source program behind the example is great details.

The insight obtained in this section is summarized as follows.

In case a reference is passed as an argument to a value parameter, the referenced object can be modified through the formal parameter

Exercise 5.3. *Passing references as ref parameters*

It is recommended that you use the web edition of the material when you solve this exercise. The web edition has direct links to the class source files, which you should use as the starting point.

In the `Date` and `Person` classes of the corresponding slide we pass a reference as *a value parameter* to method `DayDifference` in class `Date`. Be sure to understand this. Read about **ref** parameters later in this lecture.

Assume in this exercise that the formal parameter `other` in `Date.DayDifference` is passed by reference (as a C# **ref** parameter). Similarly, the actual parameter `dateOfBirth` to `DayDifference` should (of course) be passed by reference (using the keyword `ref` in front of the actual parameter).

What will be the difference caused by this program modification.

Test-drive the program with a suitable client to verify your answer.

20.5. Passing structs as value parameters

Lecture 5 - slide 22

This section is parallel to Section 20.4. In this section we pass a struct value (as opposed to an instance of a class) as a value parameter. Our finding is that a struct value (the birthday value of type `Date`) cannot be mutated from the `DayDifference` method of struct `Date`.

If we assume that `Date` is a struct instead of a class, the expression

```
  someDate.DayDifference(otherDate)
```

passes a copy of `otherDate` to `DayDifference`. The copy is discarded when we return from `DayDifference`. If `DayDifference` mutates the value of its parameter, only the local copy is affected. The value of `otherDate` is not!

In the web-version we will discuss the same example as in the web-version of Section 20.4.

Notice the following observation.

There is a good fit between use of value types and call-by-value parameter passing

If you wish the best possible fit (and no surprises) you should use value parameters with value types. The use of struct `Date` instead of class `Date` also alleviates the privacy leak problem, as pointed out in Section 16.5. See also Exercise 4.3.

Exercise 5.4. *Passing struct values as ref parameters*

This exercise corresponds to the similar exercise on the previous slide.

In the `Date` struct and the `Person` class of this slide we pass a struct value as a *value parameter* to the method `DayDifference`.

Assume in this exercise that the formal parameter `other` in `Date.DayDifference` is passed by reference (a C# **ref** parameter). Similarly, the actual parameter `dateOfBirth` to `DayDifference` should (of course) be passed by reference (using the keyword `ref` in front of the actual parameter).

What will be the difference caused by this program modification. You should compare with the version on on the slide.

Test-drive the program with a suitable client to verify your answer.

20.6. Reference Parameters

Lecture 5 - slide 23

In C, call-by-reference parameters are obtained by passing pointers as value parameters. Reference parameters in C# are **not** the same as call-by-reference parameters in C.

Reference parameters in C# are modeled after **var** parameters in Pascal. Stated briefly, a formal reference parameter in C# is an alias of the corresponding actual parameter. Therefore, the actual parameter must be a variable.

Reference parameters can be used for both input to and output from methods

Reference parameters can be used to establish alternative names (aliases) of already existing variables. The alternative names are used as formal parameters. Once established, such parameters can be used for both input and output purposes relative to a method call. The established aliases exist until the method returns to its caller.

If we - in C# - only are interested in using reference parameters for output purposes we should use **out** parameters, see Section 20.7.

Reference parameters work in the following way:

- The corresponding argument must be a variable, and it must have a value
- The types of the formal parameter and the argument must be identical
- The formal parameter becomes another name (an alias) of the argument

- The actual parameter must be prefixed with the keyword **`ref`**

In the first item it is stated that an actual reference parameter (which is a variable) must have a value before it is passed. In C#, this is called *definite assignment*.

As described in the fourth item, and as a novel contribution of C#, it is necessary to mark both formal and actual parameter with the **ref** keyword. In most other languages, only the formal parameter is marked. This may seem to be a little detail, but it implies that it is easy to spot reference parameters in a method calling form. This is very useful.

We show an example of reference parameters in Program 20.10: Swapping the values of two variables. This is the example used over and over, when reference parameters are explained.

```
using System;

public class A{
  private int a, b, c;

  public A(){
    a = 1; b = 2; c = 3;
  }

  public void Swap(ref int v1, ref int v2){
    int temp;
    temp = v1; v1 = v2; v2 = temp;
  }

  public override string ToString(){
    return String.Format("{0} {1} {2}", a, b, c);
  }

  public void Go(){
    Console.WriteLine("{0}", this);
    Swap(ref a, ref b); Swap(ref b, ref c);
    Console.WriteLine("{0}", this);
  }

  public static void Main(){
    new A().Go();
  }
}
```

Program 20.10 *The class A with a Swap method.*

In Program 20.10 we instantiate the class itself (class `A`) in the `Main` method. We send the parameterless message `Go` to this object. Hereby we take the transition from a "static situation" to an "object situation". Without this transition it would not have been possible to use the instance variables `a`, `b`, and `c` in class `A`. (They should instead have been static variables). The `Go` method pattern illustrated here is inspired from [Bishop04].

```
1 2 3
2 3 1
```

Listing 20.11 *Output of class A with Swap.*

It seems natural to support more than just value parameters in an ambitious, real-world programming language. But it is worth a consideration how much - and in which situations - to use it. We will discuss this in some details in Section 20.8.

20.7. Output Parameters

Lecture 5 - slide 24

Output parameters in C# are reference parameters used only for output purposes.

> Output parameters are used for output from methods. The method is supposed to assign values to output parameters.

Here follows the detailed rules of output parameter passing:

- The corresponding argument must be a variable
- The corresponding argument needs not to have a value on beforehand
- The formal parameter should/must be assigned by the method
- The formal parameter becomes another name (an alias) of the argument
- The types of the formal parameter and the argument must be identical
- The actual parameter must be prefixed with the keyword **`out`**

Notice the second item: It is not necessary that the actual parameter has a value before the call. In fact, the purpose of the **out** parameter is exactly to (re)initialize the actual **out** parameters. The method must ensure that the output parameter has a value (is definitely assigned) when it returns.

In Program 20.12 `DoAdd` returns the sum of the parameters `v1`, `v2`, and `v3` in the last parameter `v`. The corresponding actual parameter `r` is initialized by the call to `DoAdd` in line 21 of Program 20.12. I wrote `DoAdd` to demonstrate output parameters. Had it not been for this purpose, I would have returned the sum from `DoAdd`. In that way `DoAdd` should be called as `result = DoAdd(v1, v2, v3)`. In this case I would de-emphasize the imperative nature by calling it just `Add`. These changes describe a transition from an imperative to a more functional programming style.

```
1  using System;
2  
3  public class A{
4    private int a, b, c;
5    private int r;
6  
7    public A(){
8      a = 1; b = 2; c = 3;
9    }
10 
11   public void DoAdd(int v1, int v2, int v3, out int v){
12     v = v1 + v2 + v3;
13   }
14 
15   public override string ToString(){
16     return String.Format("{0} {1} {2}. {3}", a, b, c, r);
17   }
18 
```

```
  public void Go(){
    Console.WriteLine("{0}", this);
    DoAdd(a, b, c, out r);
    Console.WriteLine("{0}", this);
  }

  public static void Main(){
    new A().Go();
  }
}
```

Program 20.12 *The class A with a DoAdd method.*

```
1 2 3. 0
1 2 3. 6
```

Listing 20.13 *Output of class A with DoAdd.*

In Program 20.14 of Section 20.9 we show a variant of Program 20.12, which allows for an arbitrary number of actual parameters. This variant of the program is introduced with the purpose of illustrating parameter arrays.

20.8. Use of ref and out parameters in OOP

Lecture 5 - slide 25

It is interesting to wonder about the fit between object-oriented programming and the use of reference parameters. Therefore the following question is relevant.

How useful are reference and output parameters in object-oriented programming?

Output parameters are useful in case we program a method which need to produce two or more pieces of output. In such a situation, we face the following possibilities:

- Use a number of **`out`** parameters
- Mutate objects passed by value parameters
- Aggregate the pieces of output in an object and return it
- Assign the output to instance variables which subsequently can be accessed by the caller

Let us first face the first item. If a (public) method needs to pass back (to its caller) more than one piece of output, which are only loosely related to each other, it may be the best solution to use one or more out parameters for these. It should be considered to pass one of the results back via `return`.

In a language with **`ref`** and **`out`** parameters it is confusing to pass results out of method via references passed by value (call-by-value). Use a **`ref`** or an **`out`** parameter!

If the pieces of output are related - if they together form a concept - it may be reasonable to aggregate the pieces of output in a new struct or a new class. An instance of the new type can then be returned via **`return`**.

Assignment of multiple pieces of output to instance variables in the enclosing class, and subsequent access of these via properties, may compromise the original idea of the class to which the method belongs. It may also pollute the class. Therefore, it should be avoided.

`ref` and **`out`** parameters are relatively rare in the C# standard libraries

In summary, we see that there are several alternatives to the use of reference and output parameters in an object-oriented context.

Reference (**`ref`**) and output (**`out`**) parameters are only used in very few methods in the .NET framework libraries. In general, it seems to be the case that reference parameters and output parameters are not central to object-oriented programming.

20.9. Parameter Arrays

Lecture 5 - slide 26

In the previous sections we have discussed various parameter passing techniques. They were all related to the meaning of the formal parameter relative to the corresponding actual parameter (and the argument derived from the actual parameter).

In this section we will concentrate on the *parameter correspondence* mechanism. In the parameter passing discussed until now there has been a one-to-one correspondence between formal parameters and actual parameters. In this section we will study a parameter passing technique where zero, one, or more actual parameters correspond to a single formal parameter.

A parameter array is a formal parameter of array type which can absorb zero, one or more actual parameters

A formal parameter list in C# starts with a number of value/reference/output parameters for which there must exist corresponding actual parameters in a method activation. Following these parameters there can be a single parameter array, which collects all remaining arguments in an array.

With parameter arrays, there can be arbitrary many actual parameters following the 'ordinary parameters', but not arbitrary few. There must always be so many actual parameters that all the 'required' formal parameters (before a possible parameter arrays) are associated.

The following rules pertain to use of parameter arrays in C#:

- Elements of a parameter array are passed by value
- A parameter array must be the last parameter in the formal parameter list
- The formal parameter must be a single dimensional array
- Arguments that follow ordinary value, ref and out parameters are put into a newly allocated array object
- Arguments are implicitly converted to the element type of the array

It is easiest to understand parameter arrays from an example, such as Program 20.14. This is a variant of Program 20.12, which we have already discussed in Section 20.7. The `DoAdd` method takes one required parameter (which is an output parameter). As the actual out parameter in line 24, 27, and 30 we use the instance variable `r`. Following this parameter comes the parameter array called `iv` (for input values). All actual parameters after the first one must be of type `int`, and they are collected and inserted into an integer array, and made available to `DoAdd` as the `int` array `iv`.

In Program 20.14 the `DoAdd` messages to the current object add various combinations of the instance variables `a`, `b`, and `c` together. The result is assigned to the out parameter of `DoAdd`.

```
1  using System;
2  
3  public class A{
4    private int a, b, c;
5    private int r;
6  
7    public A(){
8      a = 1; b = 2; c = 3;
9    }
10 
11   public void DoAdd(out int v, params int[] iv){
12     v = 0;
13     foreach(int i in iv)
14       v += i;
15   }
16 
17   public override string ToString(){
18     return String.Format("{0} {1} {2}. {3}", a, b, c, r);
19   }
20 
21   public void Go(){
22     Console.WriteLine("{0}", this);
23 
24     DoAdd(out r, a, b, c);
25     Console.WriteLine("{0}", this);
26 
27     DoAdd(out r, a, b, c, a, b, c);
28     Console.WriteLine("{0}", this);
29 
30     DoAdd(out r);
31     Console.WriteLine("{0}", this);
32   }
33 
34   public static void Main(){
35     new A().Go();
36   }
37 }
```

Program 20.14 *The class A with a DoAdd method - both out and params.*

The output of Program 20.14 is shown in Listing 20.15. We have emphasized the value of `r` after each `DoAdd` message. Be sure that you are able to understand the parameter passing details.

```
1 2 3. 0
1 2 3. 6
1 2 3. 12
1 2 3. 0
```

Listing 20.15 *Output of class A with DoAdd - both out and params.*

The type constraint on the actual parameters can be 'removed' by having a formal parameter array of type `Object[]`.

When we studied nullable types in Section 14.9 we encountered the `IntSequence` class. In line 5 of Program 14.17 you can see a parameter array of the constructor. Please notice the flexibility it gives in the construction of new `IntSequence` objects. See also Program 14.19 where `IntSequence` is instantiated with use of the mentioned constructor.

20.10. Extension Methods

Lecture 5 - slide 27

In C#3.0 an extension method defines an instance method in an existing class without altering the definition of the class. The use of extension methods is convenient if you do not have access to the source code of the class, in which we want to have a new instance method.

Let us look at an example in order to illustrate the mechanisms behind the class extension. We use Program 11.3 as the starting point. In line 26-31 of Program 11.3 it can be observed that we have inlined calculation of distances between pairs of points. It would embellish the program if we had called `p.DistanceTo(q)` instead of

```
Math.Sqrt((p.x - q.x) * (p.x - q.x) +
          (p.y - q.y) * (p.y - q.y));
```

to find the distance between to points `p` and `q`. We will now assume that the instance method `DistanceTo` can be used on an instance of class `Point`. Program 20.16 below shows the embellishment of Program 11.3.

```
// A client of Point that instantiates three points and calculates
// the circumference of the implied triangle.

using System;

public class Application{

  public static void Main(){

    Point  p1 = PromptPoint("Enter first point"),
           p2 = PromptPoint("Enter second point"),
           p3 = PromptPoint("Enter third point");

    double p1p2Dist = p1.DistanceTo(p2),
           p2p3Dist = p2.DistanceTo(p3),
           p3p1Dist = p3.DistanceTo(p1);

    double circumference = p1p2Dist + p2p3Dist + p3p1Dist;

    Console.WriteLine("Circumference: {0} {1} {2}: {3}",
                       p1, p2, p3, circumference);
  }

  public static Point PromptPoint(string prompt){
    Console.WriteLine(prompt);
    double x = double.Parse(Console.ReadLine()),
           y = double.Parse(Console.ReadLine());
    return new Point(x,y, Point.PointRepresentation.Rectangular);
  }

}
```

Program 20.16 *A Client of class Point which uses an extension method DistanceTo.*

It is possible to extend class `Point` with the instance method `DistanceTo` without altering the source code of class `Point`. In a static class, such as in class `PointExtensions` shown below in Program 20.17, we define the `DistanceTo` method. It is defined as a static method with a first parameter prefixed with the keyword **`this`**. The C#3.0 compiler is able to translate an expression like `q.DistanceTo(q)` to `PointExtensions.DistanceTo(p,q)`. This is the noteworthy "trick" behind extension methods.

```
using System;

public static class PointExtensions{

  public static double DistanceTo(this Point p1, Point p2){
    return Math.Sqrt((p1.X - p2.X) * (p1.X - p2.X) +
                     (p1.Y - p2.Y) * (p1.Y - p2.Y));
  }

}
```

Program 20.17 *The static class PointExtensions.*

In summary, an extension method

- extends the type of the first parameter
- is defined as a static method with a `this` modifier on the first parameter
- must be defined in a static class
- cannot access private instance variables in the extended class
- is called by - behind the scene - translating to a call of the static method

Exercise 5.5. *Extending struct Double*

At the accompanying page we have seen how class `Point` can be extended externally with the method `DistanceTo`. This is an *extension method*.

If you study the method `DistanceTo` you will see that we use the squareroot function `Math.Sqrt`, defined statically in class `Math`. And we could/should have used a similar `Square` function had it been available.

It is remarkable that C# 3.0 allows us to extend the structs behind the primitive types, such as `Double` and `Int32`.

Write a static class that extends struct `Double` with the two extension methods `Sqrt` and `Square`. Afterwards, rewrite the method `DistanceTo` such that it makes use of the new instance methods in struct `Double`.

20.11. Methods versus Properties versus Indexers

Lecture 5 - slide 28

Here at the end of the chapter about methods we will summarize some rule of thumbs for choosing between properties, indexers and methods in a class:

- Properties
    - For reading and extracting of individual instance/class variables
    - For writing and assigning individual instance/class variables
    - For other kinds of data access that does not involve time consuming computations
- Indexers
    - Like properties
    - Used when it is natural to access data by indexes - *array notation* - instead of simple names
    - Used as surface notation for associative arrays
- Methods
    - For all other operations that encapsulate calculations on the data of the class

20.12. References

[Bishop04] Judith Bishop and Nigel Horspool, *C# Concisely*. Pearson. Addison Wesley, 2004.