Classes and LINQ

26. september 2017

Agenda

Classes in C#

Specialization of Classes

Inheritance in C#

LINQ

oxygen

Classes in C#

An example

Simple initial introduction to classes in object oriented programming

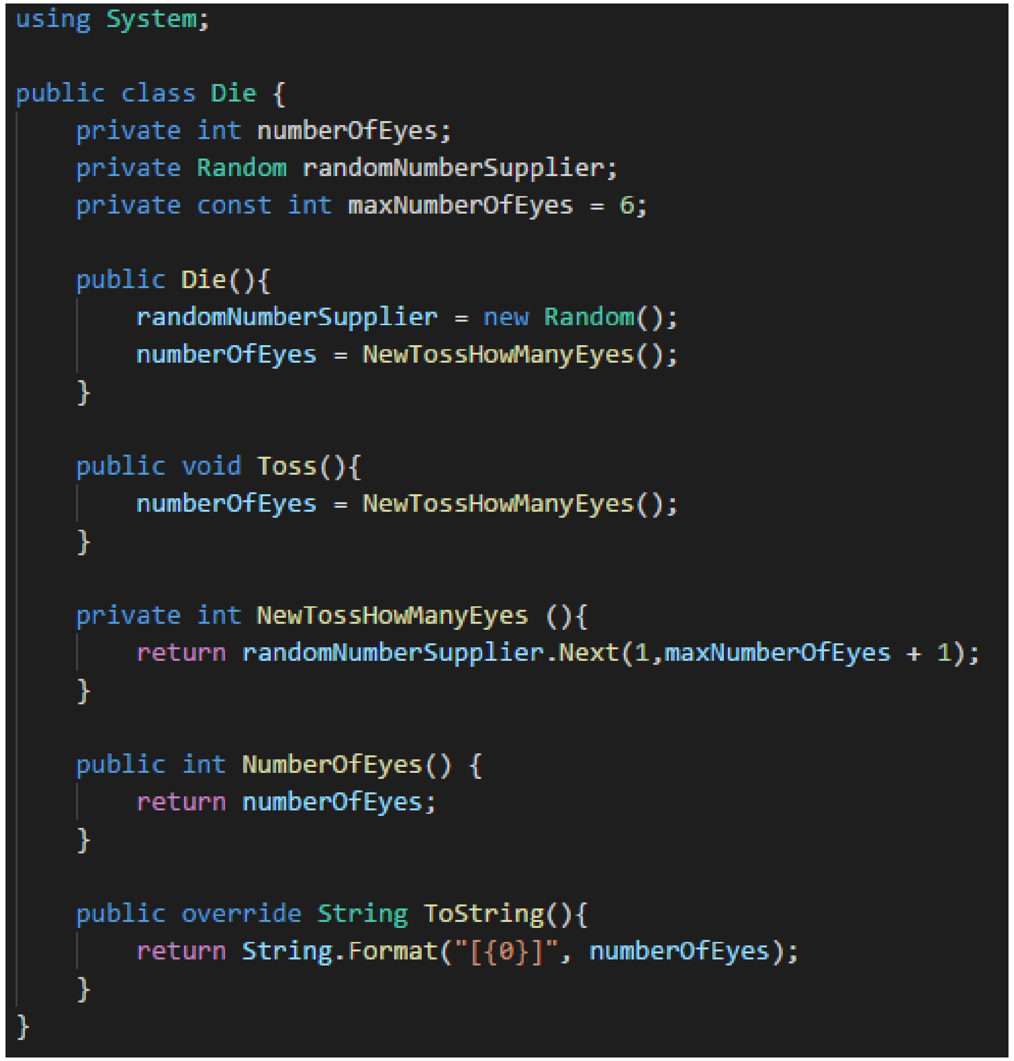
The encapsulates the instance variables

Instance variables describes the state of the class

numberOfEyes is the most important

randomNumberSupplier allows us to roll the die.

Suggestions for improvements?



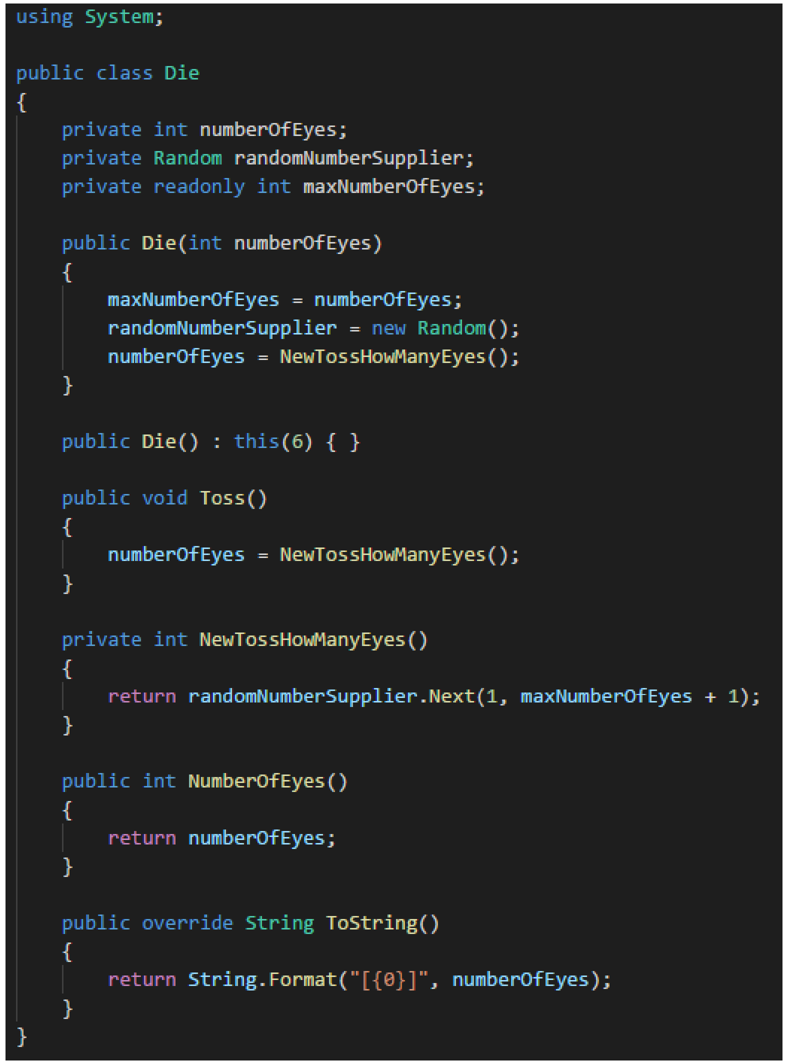
```
using System;

public class Die {
    private int numberOfEyes;
    private Random randomNumberSupplier;
    private const int maxNumberOfEyes = 6;

    public Die(){
        randomNumberSupplier = new Random();
        numberOfEyes = NewTossHowManyEyes();
    }

    public void Toss(){
        numberOfEyes = NewTossHowManyEyes();
    }

    private int NewTossHowManyEyes (){
        return randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
    }

    public int NumberOfEyes() {
        return numberOfEyes;
    }

    public override String ToString(){
        return String.Format("[{0}]", numberOfEyes);
    }
}
```

Improved for reusability

Made more reusable

Can be initialized to any size, not only as a 6-sided die

It defaults to a 6-sided die



```
using System;

public class Die
{
    private int numberOfEyes;
    private Random randomNumberSupplier;
    private readonly int maxNumberOfEyes;

    public Die(int numberOfEyes)
    {
        maxNumberOfEyes = numberOfEyes;
        randomNumberSupplier = new Random();
        numberOfEyes = NewTossHowManyEyes();
    }

    public Die() : this(6) { }

    public void Toss()
    {
        numberOfEyes = NewTossHowManyEyes();
    }

    private int NewTossHowManyEyes()
    {
        return randomNumberSupplier.Next(1, maxNumberOfEyes + 1);
    }

    public int NumberOfEyes()
    {
        return numberOfEyes;
    }

    public override String ToString()
    {
        return String.Format("[{0}]", numberOfEyes);
    }
}
```

Encapsulation

The single most important aspect of a class is encapsulation

A class is a construct that surrounds a number of definitions, which belong together

Some of these definitions can be seen from the outside, whereas others are only relevant seen from the inside

A class encapsulates data and operations that belong together, and it controls the visibility of both data and operations. A class can be used as a type in the programming language

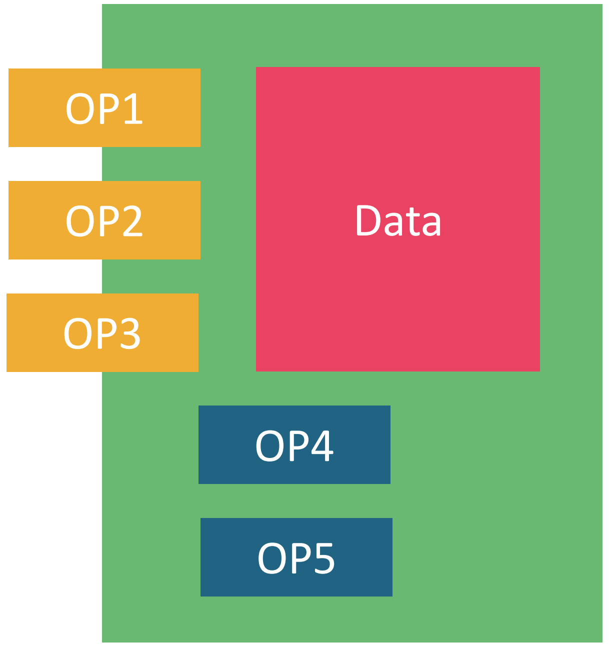
Encapsulation (cont.)

The parts of a class which are visible from other classes forms the *client interface* of the class.

All data parts are kept inside the class, and they cannot be directly used from other classes.

The notion of interfaces between program parts - program building blocks - is important in general

It turns out that a class may have several different interfaces



Exercise – Time class

Design a class **PointInTime**, which represents a single point in time.

- How do we represent a point in time?
- Which variables (data) should be encapsulated in the class?
  - Design the variables in terms of their names and types.
- Which operations should constitute the client interface of the class?
  - Design the operations in terms of their names, formal parameters (and their types), and the types of their return values

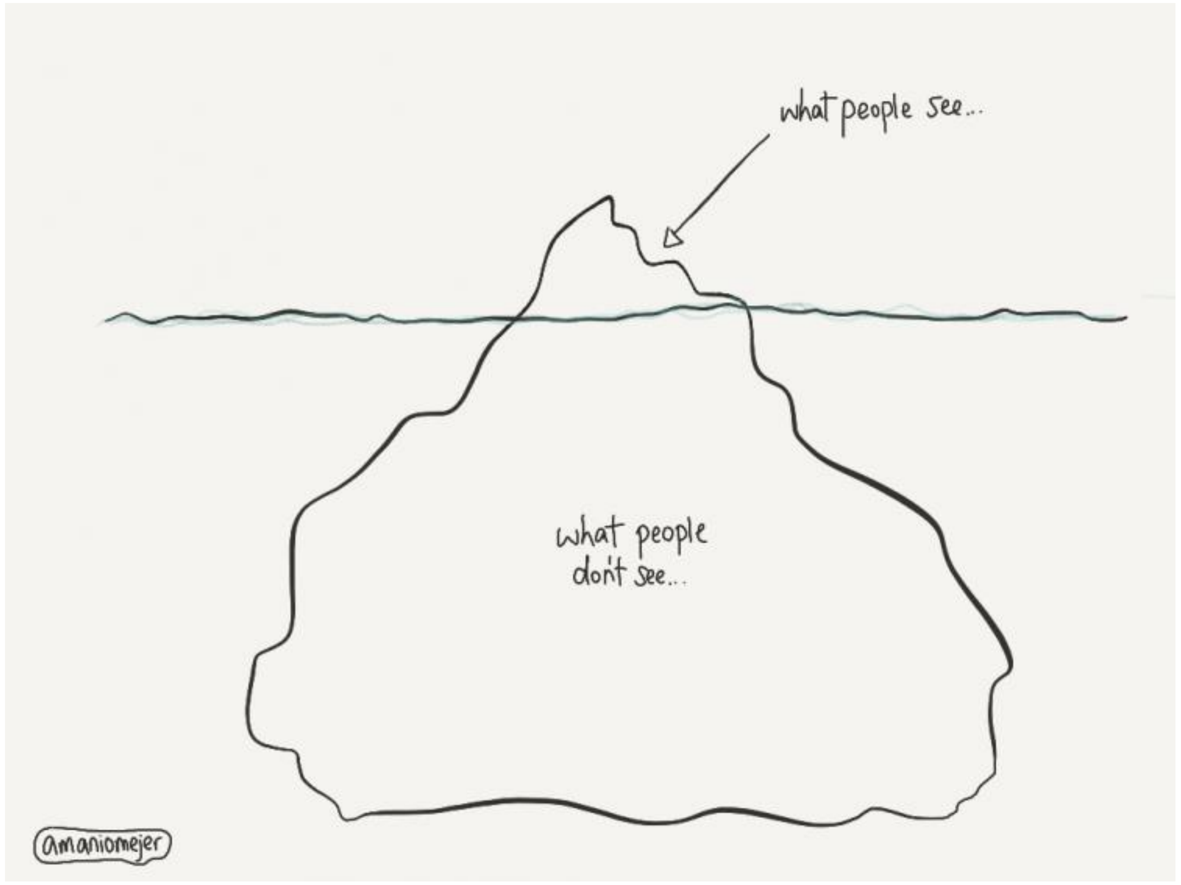
Avoid looking at the time-related types in the C# library before you solve this exercise!

Visibility – the iceberg analogy

A class can be seen as an iceberg:
Only a minor part of it should be visible from the outside. The majority of the class details should be hidden.

Clients of a class C cannot directly depend on hidden parts of C.

Thus, the invisible parts in C can more easily be changed than the parts which constitute the interface of the class.



Visibility (cont.)

Visibility-control is important because it protects the invisible parts of a class from being directly accessed from other classes.

No other parts of a program can rely directly on details, which they cannot see/access.

If some detail (typically a variable) of a class cannot be seen outside the class, it is much easier to modify this detail (e.g. replace the variable by a set of other variables) at a later point in time.

Visible and hidden aspects

Visible aspects

- The name of the class
- The signatures of selected operations: The interface of the class

Hidden aspects

- The representation of data
- The bodies of operations
- Operations that solely serve as helpers of other operations

The visible aspects should be kept at minimum level

It is always recommended to keep the representation of data secret

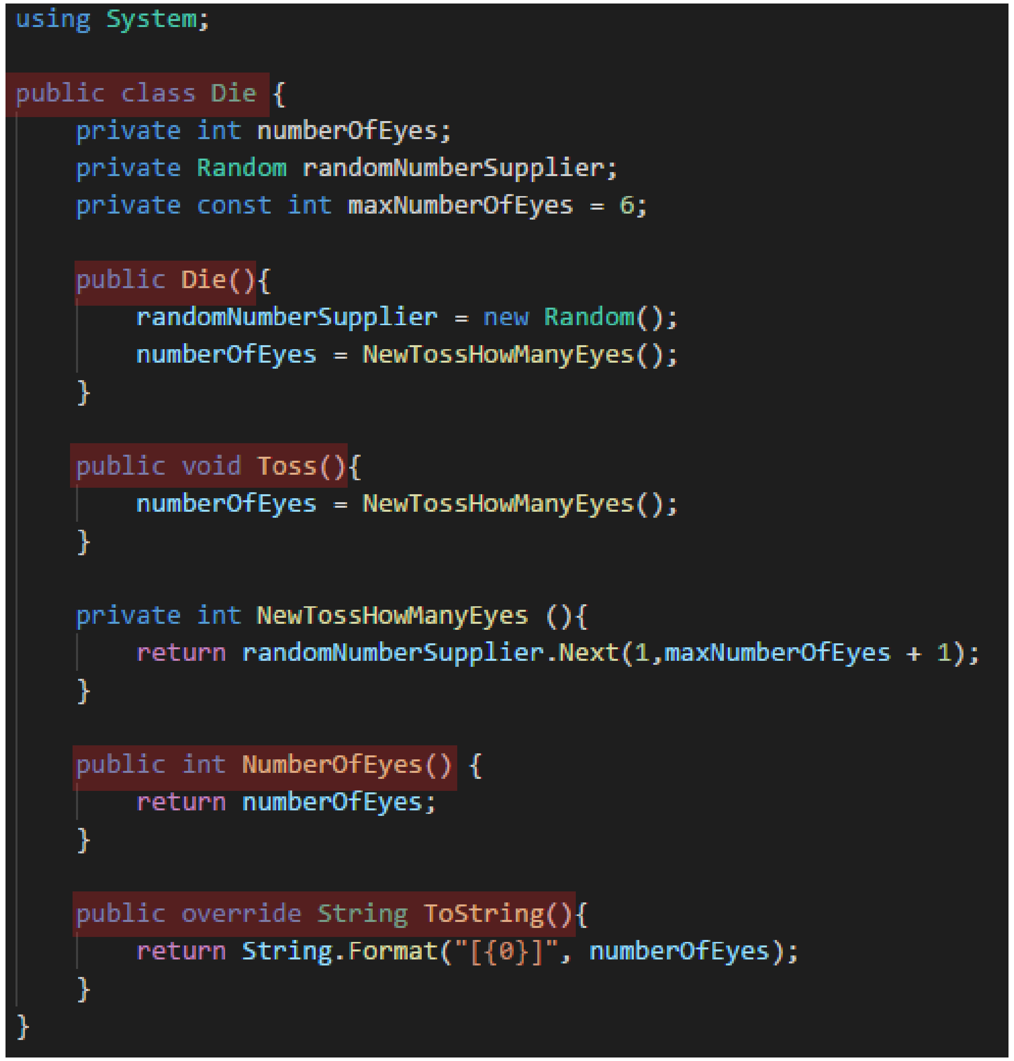
Visible and hidden aspects (cont.)

The image shows the visible aspects (🟥)

Some programming language enforce that all instance variables of a class are hidden.

Smalltalk is one such language.

C# is not in this category, but we will typically strive for such discipline in the way we program in C#.



```
using System;

public class Die {
    private int numberOfEyes;
    private Random randomNumberSupplier;
    private const int maxNumberOfEyes = 6;

    public Die(){
        randomNumberSupplier = new Random();
        numberOfEyes = NewTossHowManyEyes();
    }

    public void Toss(){
        numberOfEyes = NewTossHowManyEyes();
    }

    private int NewTossHowManyEyes (){
        return randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
    }

    public int NumberOfEyes() {
        return numberOfEyes;
    }

    public override String ToString(){
        return String.Format("[{0}]", numberOfEyes);
    }
}
```

Program modification – fire analogy

A minor modification may spread as fire throughout the program

Such a fire may ruin most of the program in the sense that major parts of the program may need to be reprogrammed.

The use of firewalls prevents the spread of a fire.

Similarly, encapsulation and visibility control prevent program modifications from having global consequences.

Representation of independence

An important Object-oriented programming principle:

Representation independence:

Clients of the class ***C*** should not be affected by changes of ***C***'s data representation

In essence, this is the idea we have already discussed. Now we have name for it!

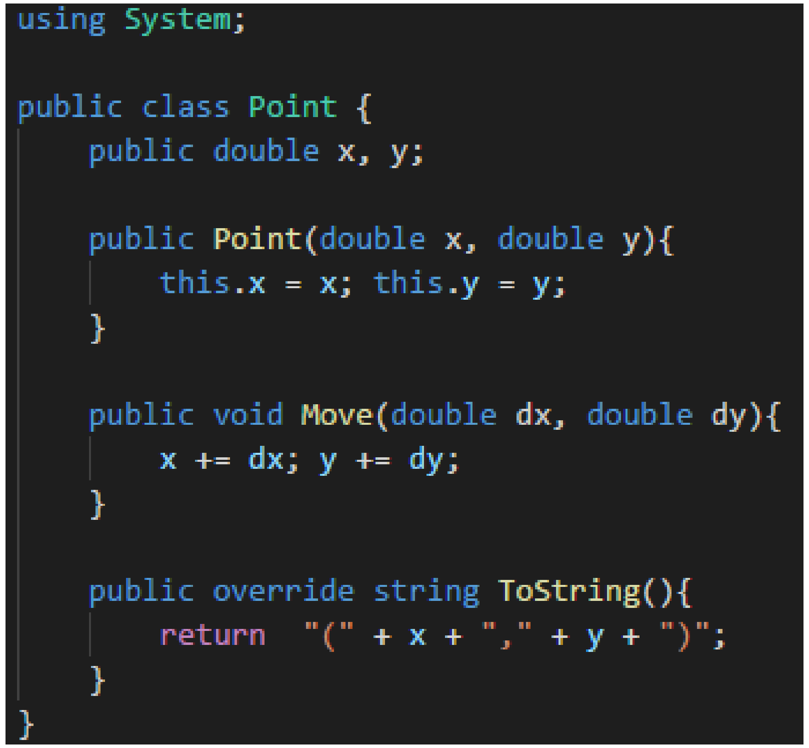
Representation of independence (cont.)

A program using *x* and *y* is utterly dependent upon the specific implementation

If for some reason the implementation of Point changes, the dependent program will break

If we change the Point class to use radius and angle instead.

Encapsulated data should always be *hidden* and *private* within the class



```
using System;

public class Point {
    public double x, y;

    public Point(double x, double y){
        this.x = x; this.y = y;
    }

    public void Move(double dx, double dy){
        x += dx; y += dy;
    }

    public override string ToString(){
        return  "(" + x + "," + y + ")";
    }
}
```

Classes in C#

The default visibility of members in a class is private.

It means that if you do not provide a visibility modifier of a variable or a method, the variable or method will be private.

This is unfortunate, because a missing visibility modifier typically signals that the programmer forgot to decide the visibility of the member.

It would have been better design of C# to get a compilation error or - at least - a warning.

Classes in C# (cont.)

The following gives an overview of different kinds of members - variables and methods - in a class:

- Instance variable
  - Defines state that is related to each individual object
- Class variable
  - Defines state that is shared between all objects
- Instance method
  - Activated on an object. Can access both instance and class variables
- Class method
  - Accessed via the class. Can only access class variables

Instance variables

All objects of a particular class have the same set of variables.

Each object allocates enough memory space to hold its own set of variables.

Thus, the values of these variables may vary from one instance (object) to another.

Therefore the variables are known as *instance variables*.

Unfortunately, the terminology varies a lot.

Instance variables are officially known as *fields* in C#.

Together with constants, these are known as data members

The term *member* is often used for all declarations contained in a class

Exercise – Modify Bank Account

Given the BankAccount class.

Now modify this class such that each bank account has a backup account.

For the backup account you will need a new (private) instance variable of type BankAccount.

Modify the Withdraw method, such that if there is not enough money available in the current account, then withdraw the money from the backup account.

As an experiment, access the balance of the backup account directly, in the following way:

```
backupAccount.balance -= ...
```

Is it possible to modify the private state of one BankAccount from another BankAccount? Discuss and explain your findings. Are you surprised?

Instance methods

Instance methods are intended to work on (do computations on) the instance variables of an object in a class

An instance method *M* must always be activated on an instance (an object) of the class to which *M* belongs.

- An instance method *M* in a class **C**
  - must be activated on an object which is an instance of **C**
  - is activated by object.*M*(...) from outside **C**
  - is activated by this.*M*(...) or just *M*(...) inside **C**
  - can access all members of **C**

Class variables

A class variable in a class **C** is shared between all instances (objects) of **C**.

In addition, a class can be used even in the case where there does not exist any instance of C at all.

Some classes are not intended to be instantiated.

- Class variable
  - are declared by use of the static modifier in C#
  - may be used as global variables - associated with a given class
  - do typically hold meta information about the class, such as the number of instances

Class methods

Class methods are not connected to any instance of a class.

Thus, class methods can be activated without providing any instance of the class.

A class method *M* in a class **C** is activated by **C**.*M*(...).

The static method Main plays a particular role in a C# program, because the program execution starts in Main.

It is crucial that Main is static, because there are objects around at the time Main is called.

Thus, it is not possible to activate any instance method at that point in time!

We have seen Main used many times already.

Class methods (cont.)

- A class method M in a class **C**
  - is declared by use of the static modifier in C#
  - can only access static members of the class
  - must be activated on the class as such
  - is activated as **C**.*M*(...) from outside **C**
  - can also be activated as *M*(...) from inside **C**

Static and Partial classes

A static class C can only have static members

A partial class is defined in two or more source files

Static and Partial classes (cont.)

- Static class
  - Serves as a module rather than a class
  - Prevents instantiation, subclassing, instance members, and use as a type.
  - Examples: System.Math, System.IO.File, and System.IO.Directory
- Partial class
  - Usage: To combine manually authored and automatically generated class parts.

Constant and readonly variables

The variables we have seen until now can be assigned to new values at any time during the program execution.

In this section we will study variable with no or limited assignment possibilities.

Of obvious reasons, it is confusing to call these "variables". Therefore we use the term "constant" instead.

C# supports two different kinds of constants.

Some constants, denoted with the *const* modifier, are bound at compile time.

Others, denoted with the *readonly* modifier, are bound at object creation time.

Constants and readonly variables cannot be changed during program execution

Constant and readonly variables (cont.)

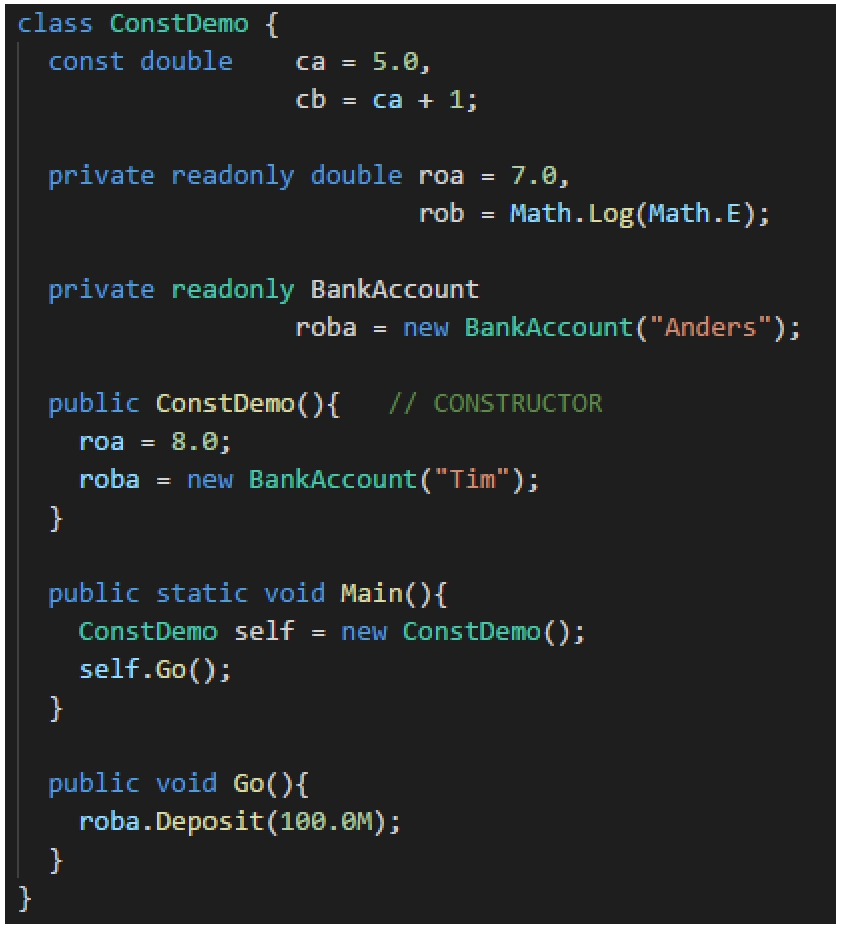
Constants declared with use of the *const* keyword

- Computed at compile-time
- Must be initialized by an initializer
- The initializer is evaluated at compile time
- No memory is allocated to constants
- Must be of a simple type, a string, or a reference type

Readonly variables declared with use of the *readonly* modifier

- Computed at object-creation time
- Must either be initialized by an initializer or in a constructor
- Cannot be modified in other parts of the program

Constant and readonly variables (cont.)



```
class ConstDemo {
  const double    ca = 5.0,
                  cb = ca + 1;

  private readonly double roa = 7.0,
                          rob = Math.Log(Math.E);

  private readonly BankAccount
                  roba = new BankAccount("Anders");

  public ConstDemo(){   // CONSTRUCTOR
    roa = 8.0;
    roba = new BankAccount("Tim");
  }

  public static void Main(){
    ConstDemo self = new ConstDemo();
    self.Go();
  }

  public void Go(){
    roba.Deposit(100.0M);
  }
}
```

Objects and classes

At an overall level objects are often characterized in terms of *identity*, *state*, and *behaviour*

An object has an *identity* which makes it different and distinct from any other object

Two objects which are created by two activations of the new operator never share identity (they are not identical)

In the practical world, the identity of an object is associated to its location in the memory: its address

The *state* of the object corresponds to the data, as prescribed by the class to which the object belongs. As such, the state pertains to the instance variables of the class

The *behaviour* of the object is prescribed by the operations of the class, to which the object belongs.

Objects and classes (cont.)

We practice *object-oriented programming*, but we write classes in our programs.

This may be a little confusing. Shouldn't we rather talk about *class-oriented programming*?

When we write an object-oriented program, we are able to program all (forthcoming) objects of a given type/class together. This is done by writing the class.

Thus, we write the classes in our source programs, but we often imagine a (forthcoming) situation where the class "is an object" which interacts with a number of other objects - of the same type or of different types.

The classes exist for a long time - typically years. In the running program we have objects. The objects exist while the program is running. A typical program runs for a short time.

Often, we want to preserve our objects in between program executions. This turns out to be a challenge!

Objects and classes (cont.)

All objects cease to exist when the program execution terminates.

This is in conflict with the behavior of corresponding real-life phenomena, and it causes a lot of problems and challenges in many programs

Classes are written and described in source programs

Objects are created and exist while programs are running

The current object - *this*

*The current object in a C# program execution is denoted by the variable **this***

this is used for several different purposes in C#:

- Reference to shadowed instance variables
- Activation of another constructor
- Definition of indexers
- In definition of extension methods

Visibility issues

Let us first summarize some facts about visibility of types and members:

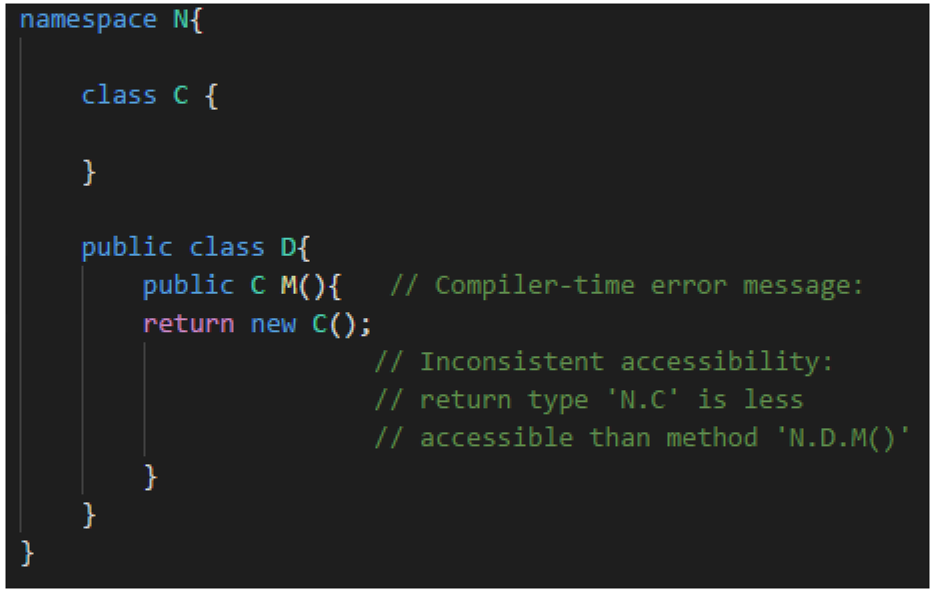
- Types in namespaces
  - Either public or internal
  - Default visibility: internal
- Members in classes
  - Either private, public, internal, protected or internal protected
  - Default visibility: private
- Visibility inconsistencies
  - A type T can be less accessible than a method that returns a value of type T

Visibility issues (cont.)

We have shown an internal class ***C*** in a namespace ***N***

C is only supposed to be used inside the namespace ***N***

In reality we have forgotten to state that ***C*** is public in ***N***


```
namespace N{

    class C {

    }

    public class D{
        public C M(){   // Compiler-time error message:
        return new C();
                      // Inconsistent accessibility:
                      // return type 'N.C' is less
                      // accessible than method 'N.D.M()'
        }
    }
}
```

oxygen

Specialization of Classes

Inheritance

Inheritance represents an organization of classes in which one class, ***B***, is defined on top of another class, ***A***.

Class ***B*** inherits the members of class ***A***, and in addition ***B*** can define its own members.

Use of inheritance makes it possible to *reuse* the data and operations of a class ***A*** in several so-called *subclasses*, such as ***B***, ***C***, and ***D***, without copying these data and operations in the source code.

Thus, if we modify class ***A*** we have also implicitly modified ***B***, ***C*** and ***D***.

Specialization of classes

Classes are regarded as types, and specializations as subtypes

Specialization facilitates definition of new classes from existing classes on a sound conceptual basis

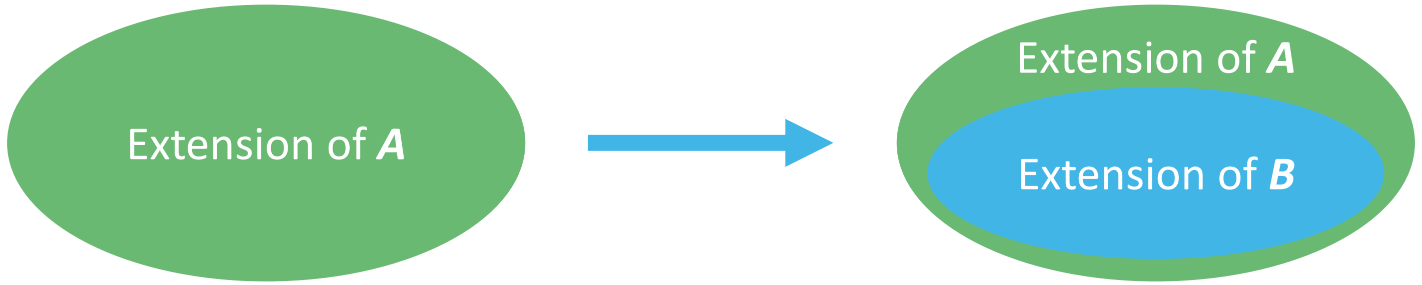
If a class ***B*** is a specialization of a class ***A*** then

- The instances of ***B*** is a subset of the instances of ***A***
- Operations and variables in ***A*** are also present in ***B***
- Some operations from ***A*** may be redefined in ***B***

Extension of class specialization

The extension of a specialized class B is a subset of the extension of the generalized class A

The relationships between the extension of ***A*** and ***B*** can be illustrated as follows, using the well-known notation of venn-diagrams.



Extension of class specialization (cont.)

Let us now introduce the is-a relation between the two classes ***A*** and ***B***:

- A ***B***-object **is an** ***A***-object
- There is a **is-a** relation between class ***A*** and ***B***

The **is-a** relation characterizes specialization. We may even formulate an "is-a test" that tests if B is a specialization of A. The **is-a** relation can be seen as contrast to the **has-a** relation, which is connected to *aggregation*

The is-a relation forms a contrast to the has-a relation

The is-a relation characterizes specialization

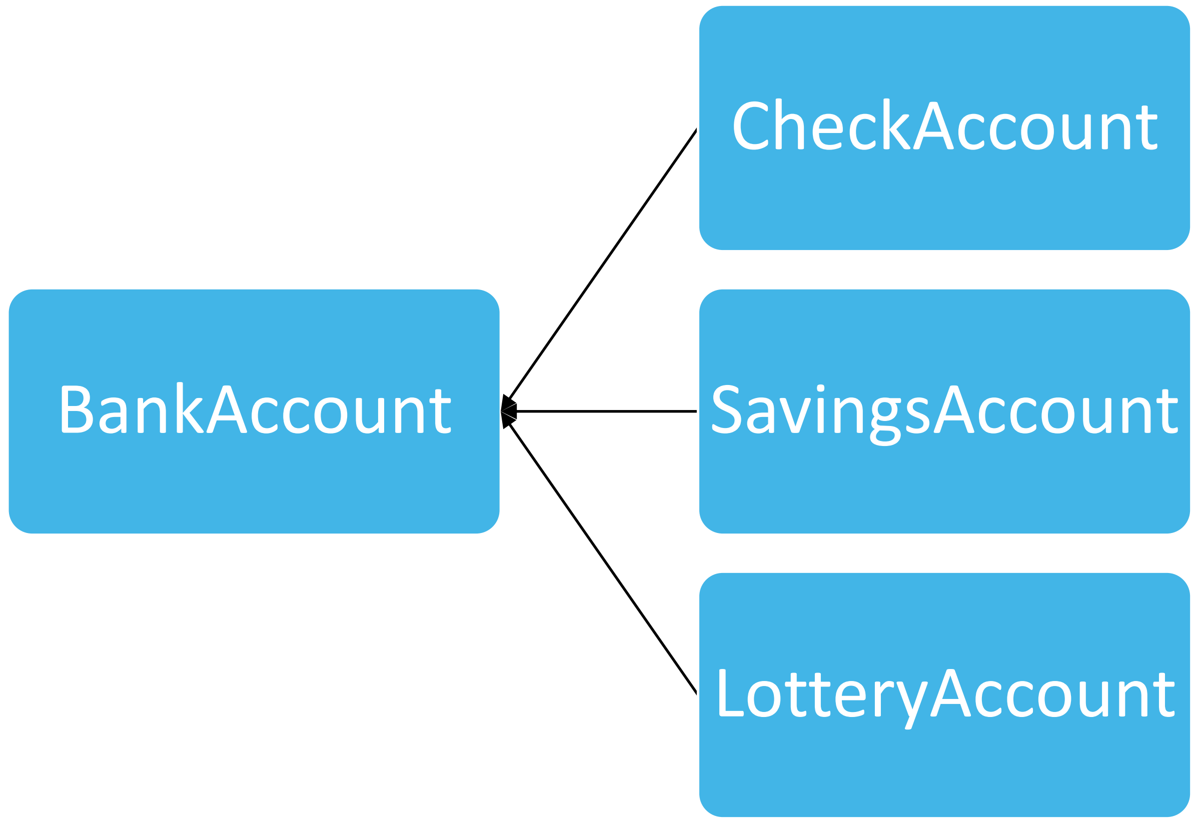
The has-a relation characterizes aggregation

Example: Bank accounts

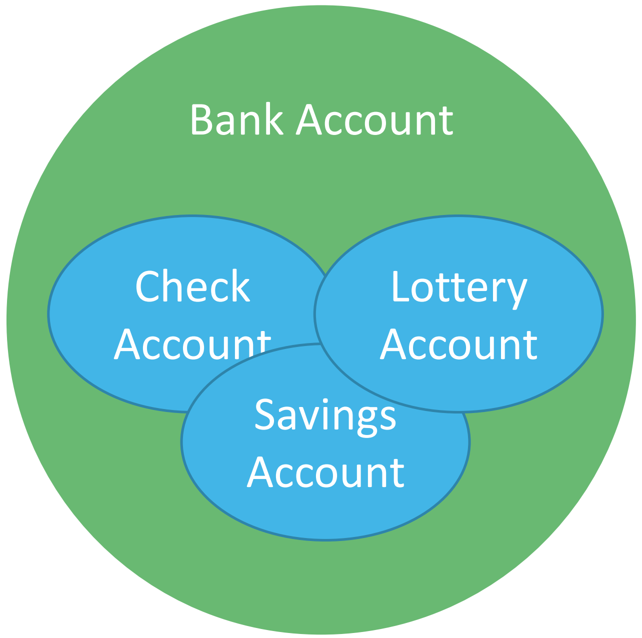
The is-a test confirms that there is a generalization-specialization relationship between BankAccount and CheckAccount:

The statement "CheckAccount is a BankAccount" captures the relationships between the two classes.

As a contrast, the has-a test fails: It is against our intuition that "a CheckAccount has a BankAccount"



Example: Bank Accounts (cont.)



Exercise: Implement a Savings account

Implement a SavingsAccount class, which inherits from the BankAccount class from earlier.

Modify BankAccount to have Virtual methods

Override Withdraw, AddInterests and ToString in SavingsAccount

Hint: SavingsAccount : BankAccount

Summarization

Objects of specialized classes

- fulfill stronger conditions (constraints) than objects of generalized classes
  - obey stronger class invariants
- have simpler and more accurate operations than objects of generalized classes

Specialization of classes in pure form do not occur very often.

Specialization in combination with extension is much more typical.

The principle of substitution

The principle of substitution is described by Timothy Budd in section 8.3 of in his book *An Introduction to Object-oriented Programming*

The principle of substitution describes an ideal, which not always is in harmony with our practical and everyday programming experience.

This corresponds to our observation that pure specialization only rarely is found in real-life programs.

*If **B** is a subclass of **A**, it is possible to substitute an given instance of **B** in place of an instance of **A** without observable effect*

The principle of substitution (cont.)

Notice that opposite substitution does not always work

Thus, we cannot substitute a triangle with a general polygon (for instance a square)

Most programs would break immediately if that was attempted

The reason is that a square does not, in general, possess the same properties as a triangle

Extension of Classes

Classes can both be regarded as types and modules

Class extension is a program transport and program reusability mechanism

Extension of classes (cont.)

As the name suggests, class extension is concerned with adding something to a class

We can add both variables and operations

We are not constrained in any way (by ideals of specialization or substitution) so we can in principle add whatever we want.

However, we still want to have coherent and cohesive classes.

We want classes focused on a single idea, where all data and operations are related to this idea.

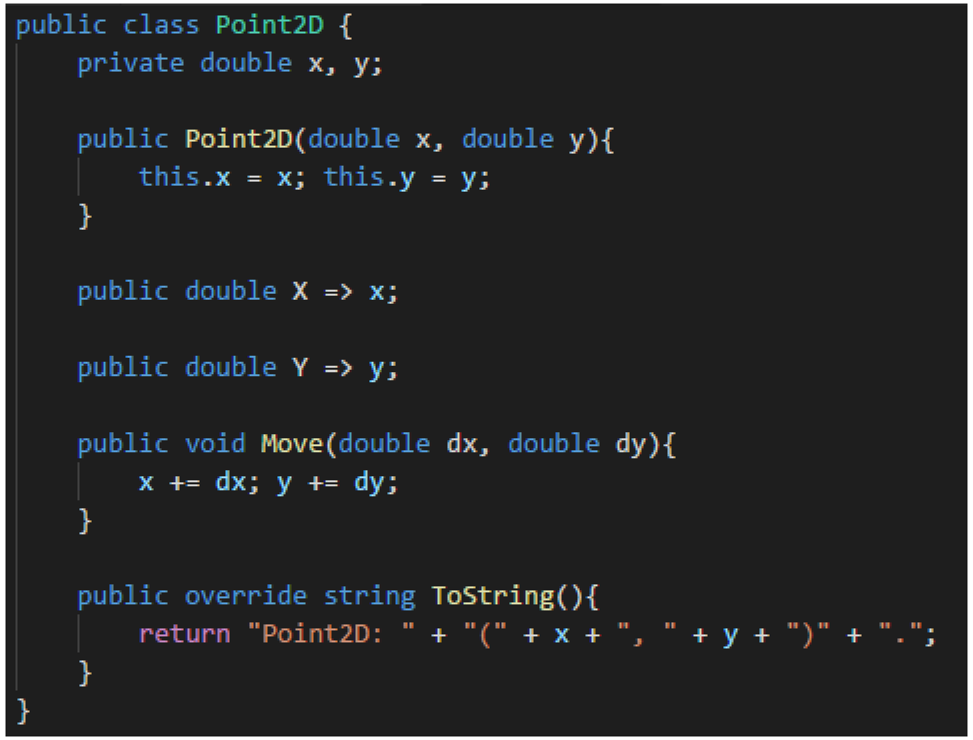
Our classes should be used as types for declaration of variables, and it should make sense to make instances of the classes.

Extension of classes (cont.)

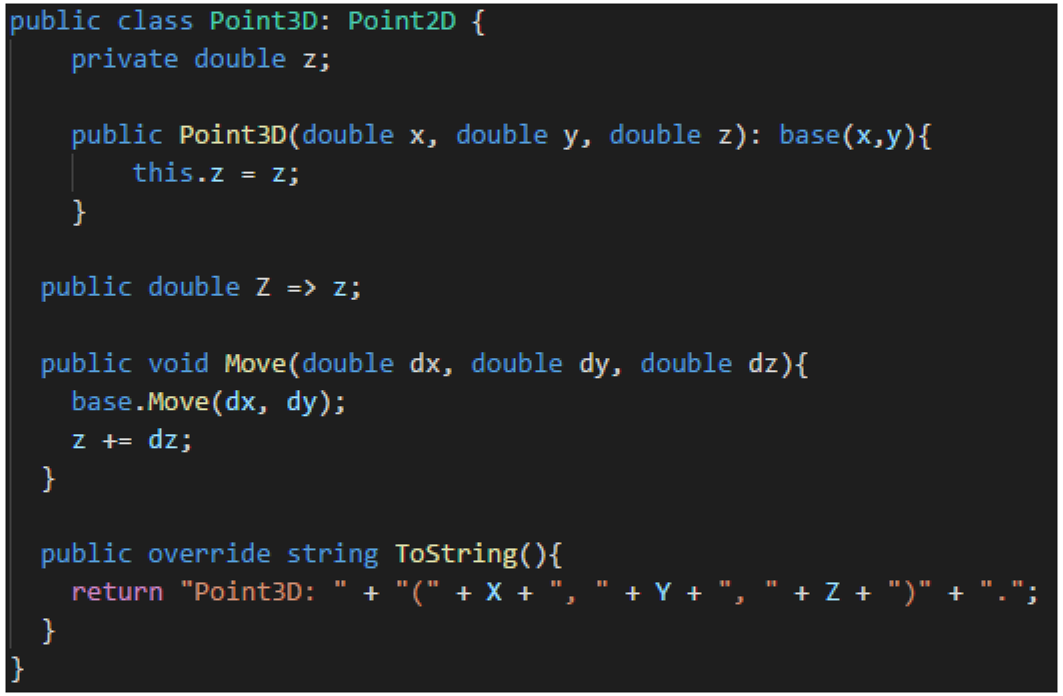
If class ***B*** is an extension of class ***A*** then

- ***B*** may add new variables and operations to ***A***
- Operations and variables in ***A*** are also present in ***B***
- ***B***-objects are not necessarily conceptually related to ***A***-objects

Example: Point2D → Point3D



```
public class Point2D {
    private double x, y;

    public Point2D(double x, double y){
        this.x = x; this.y = y;
    }

    public double X => x;

    public double Y => y;

    public void Move(double dx, double dy){
        x += dx; y += dy;
    }

    public override string ToString(){
        return "Point2D: " + "(" + x + ", " + y + ")" + ".";
    }
}
```



```
public class Point3D: Point2D {
    private double z;

    public Point3D(double x, double y, double z): base(x,y){
        this.z = z;
    }

  public double Z => z;

  public void Move(double dx, double dy, double dz){
    base.Move(dx, dy);
    z += dz;
  }

  public override string ToString(){
    return "Point3D: " + "(" + X + ", " + Y + ", " + Z + ")" + ".";
  }
}
```

The important observations about the extension Point3D of Point2D can be stated as follows:

- A 3D point is not a 2D point
- Thus, Point3D is not a specialization of Point2D
- The set of 2D point objects is disjoint from the set of 3D points

Intension of class extension

We have realized that the essential characteristics of specialization is the narrowing of the class extension.

We also realized that the class extension of an extended class (such as Point3D) typically is disjoint from the class extension of the parent class (such as Point2D).

The intension of a class extension B is a superset of the intension of the original class A

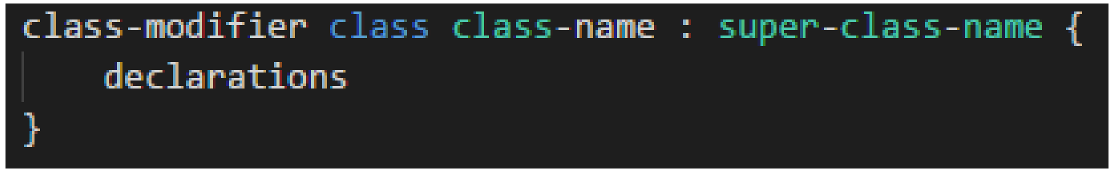
It is, in general, not possible to characterize the extension of B in relation to the extension of A

Often, the extension of A does not overlap with the extension of B

Inheritance in C#

Class Inheritance in C#

When we define a class, ***class-name***, we can give the name of the superclass, ***super-class-name***, of the class. (Superclass is sometimes called base class)



```
class-modifier class class-name : super-class-name {
    declarations
}
```

We see that the superclass name is given after the colon. There is no keyword involved (like extends in Java)

If a class implements interfaces, the names of these interfaces are also listed after the colon

The superclass name must be given before the names of interfaces

If we do not give a superclass name after the colon, it is equivalent to writing ***: Object***

A class, which does not specify an explicit superclass, inherits from class Object

Methods in the class Object in C#

Public methods in class Object

- Equals:
  - obj1.Equals(obj2) - Instance method
  - Object.Equals(obj1, obj2) - Static method
  - Object.ReferenceEquals(obj1,obj2) - Static method
- obj.GetHashCode()
- obj.GetType()
- obj.ToString()

Protected methods in class Object

- obj.Finalize()
- obj.MemberwiseClone()

Inheritance and Constructors

As the only kind of members, constructors are not inherited.

This is because a constructor is only useful in the class to which it belongs.

Here follows the overall guidelines for constructors in class hierarchy:

- Each class in a class hierarchy should have its own constructor(s)
- The constructor of class ***C*** cooperates with constructors in super classes of ***C*** to initialize a new instance of ***C***
- A constructor in a subclass will always, implicitly or explicitly, refer to a constructor in its superclass

Constructors and initialization order

A constructor in a subclass must - either implicitly or explicitly - activate a constructor in a superclass

In that way a chain of constructors are executed when an object is initialized

The chain of constructors will be called from the most general to the least general

The following initializations take place when a new ***C*** object is made with new ***C***(...):

- Instance variables in ***C*** are initialized (field initializers)
- Instance variables in super classes are initialized - most specialized first
- Constructors of the super classes are executed - most general first
- The constructor body of ***C*** is executed

Visibility modifiers in C#

Basically, we must distinguish between visibility of types in assemblies and visibility of members in types:

- Visibility of a type (e.g. a class) in an assembly
  - internal: The type is not visible from outside the assembly
  - public: The type is visible outside the assembly
- Visibility of members in type (e.g., methods in classes)
  - private: Accessible only in the containing type
  - protected: Accessible in the containing type and in subtypes
  - internal: Accessible in the assembly
  - protected internal: Accessible in the assembly and in the containing type and its subtypes
  - public: Accessible whenever the enclosing type is accessible

Methods, properties, and indexers

All members apart from constructors are inherited. In particular we notice that operations (methods, properties, and indexers) are inherited.

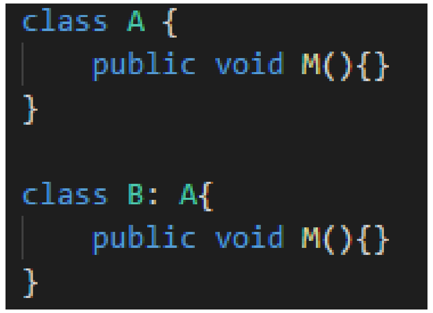
Here follows some basic observations about inheritance of operations:

- Methods, properties, and indexers can be redefined in two different senses:
    - Same names and signatures in super- and subclass, closely related meanings (virtual, override)
    - Same names and signatures in super- and subclass, two entirely different meanings (new)
- A method *M* in a subclass ***B*** can refer to a method *M* in a superclass ***A***
    - base.*M*(...)
    - Cooperation, also known as method combination

Operators are inherited. A redefined operator in a subclass will be an entirely new operator.

Overriding and Hiding in C#

Let us now carefully explore the situation where a method *M* appears in both class ***A*** and its subclass ***B***.



```
class A {
    public void M(){}
}

class B: A{
    public void M(){}
}
```

This program is illegal

There are basically two different situations that make sense:

- Intended redefinition:
  - ***B***.*M* is intended to redefine A.M - such that B.M is used on B instances
  - ***A***.*M* must be declared as virtual
  - ***B***.*M* must be declared to override A.M
- Accidental redefinition:
  - The programmer of class ***B*** is not aware of ***A***.*M*
  - ***B***.*M* must declare that it is not related to ***A***.*M* - using the new modifier

Polymorphism. Static and dynamic types

Polymorphism stands for the idea that a variable can refer to objects of several different types

The static type of a variable is the type of variable, as declared

The dynamic type of a variable is type of object to which the variable refers

Dynamic binding is in effect if the dynamic type of a variable v determines the operation activated by v.op(...)

Inheritance and Variables

Variables (fields) are inherited, variables cannot be virtual

Thus a variable v in a superclass ***A*** is present in a subclass ***B***. This is even the case if *v* is private in class ***A***

oxygen

Questions?