Programming Using C#,
Basic Course

Object-Oriented Programming with C#
Encapsulation and data hiding

Agenda:

- Classes and objects
- Encapsulation and data hiding
- Constructors
- Overloaded methods

Farid Naisan, University Lecturer, Malmö University, farid.naisan@mah.se

Classes and objects

- A class defines a group of similar objects. It is a blueprint from which one or more objects of that type may be created.
- A program in Java consists of only classes. Most programs are made of many classes.
- An object created from a class is called an instance of the class
- While a class is only a definition, an object is a universally unique “thing” that exists.
  - The class Car describes cars in general.
  - yourCar, myCar are two unique objects.

Classes and objects.

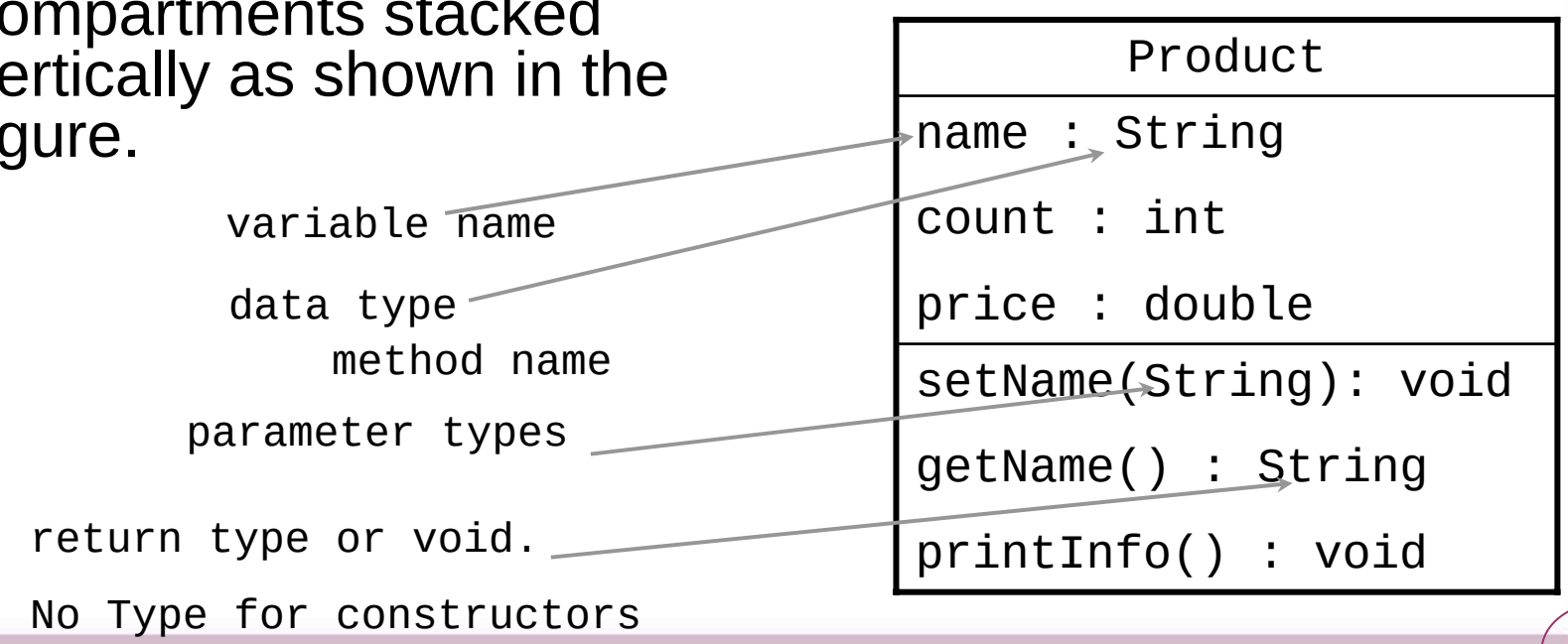
- Classes can represent physical objects like `Car, House, TV`, or conceptual objects like `Address, BankLoan,` or `Rectangle`.
- A class is a type and an object can be thought of as variable.
- Objects are created by the keyword new and placed on the heap memory.
- A reference variable contains the address of an object.

```
BankLoan loan = new BankLoan();
```

Parts of a class

- A class contains:
  - fields for storing data,
  - methods for performing tasks.
- Classes are modeled as a rectangle containing three compartments stacked vertically as shown in the figure.

Class Name
Fields (attributes)
Methods



Class declaration

- A very general layout of a class may look like this:`using System;`

```
using System;

//Other using-statements
namespace Products
{
    public class Product
    {
        //fields (instance variables
        //Constructors
        public Product()
        {
        }
        //Properties (get- and set methods)

       //other methods
    } //class
} //namespace
```

Exempel

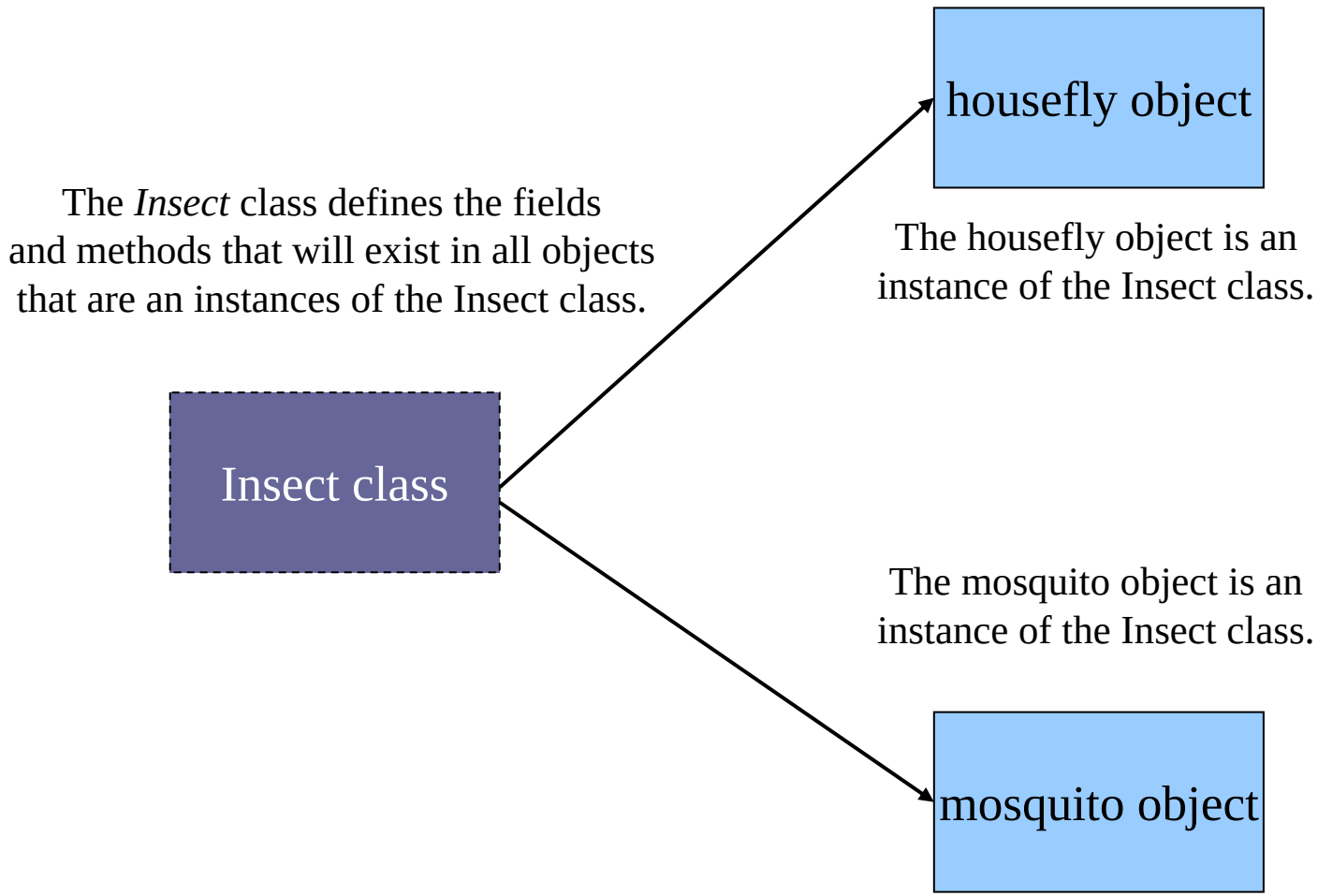
- Here is an example.
- Constructors and Properties will are discussed later in this presentation.

```
using System;
 //andra using-satser
namespace Products
{
    public class Product
    {
        //Instance fields
        string name;   //name of product
        int count = 0; //number in stock
        decimal price; //sales prices without VAT

        //Default Constructor
        public Product()
        {
            Initialize();
        }
        //Other construtors

        //Properties
        public String Name
        {
            get { return name; }
            set { name = value; }
        }
        //Mthods
        public void Initialize()
        {
            name = "NoName";
            count = 0;
            price = 0.0m;
        }
        //other methods
    }//class
}//namespace
```

Classes and Objects

- The programmer determines the fields and methods needed, and then creates a class.
- A class can specify the fields and methods that a particular type of object may have.
- A class is a “blueprint” that objects may be created from.
- A class is not an object, but it can be a description of an object.
- An object created from a class is called an *instance* of the class.

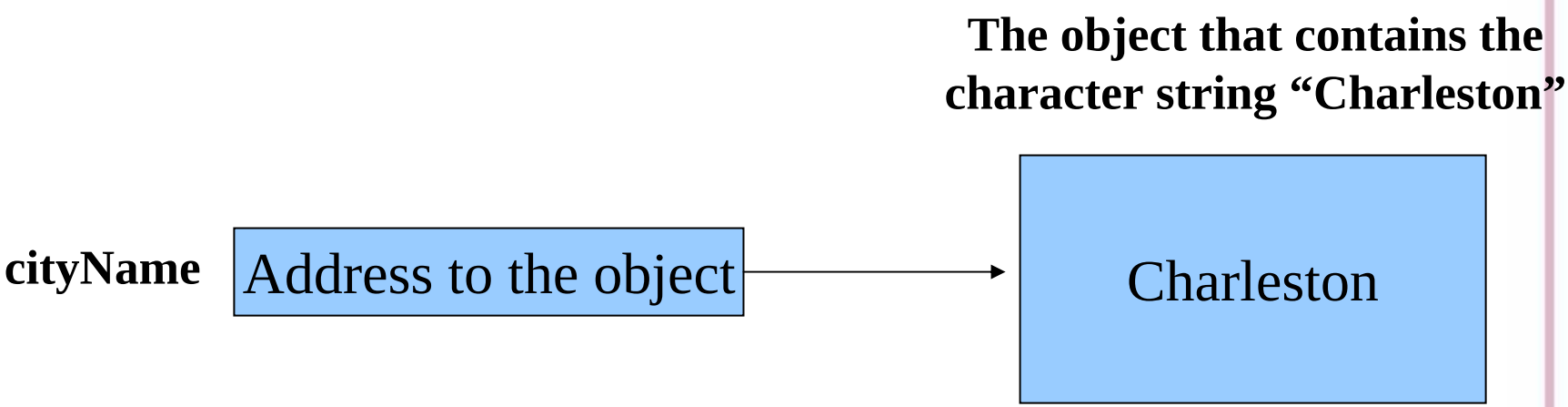
Classes and Objects



Classes

- A reference variable contains the address of an object.

```
string cityName = “Charleston”;
```



Classes

- The `Length` property of the string class returns and integer value that is equal to the length of the string.

```
int stringLength = cityName.Length;
```

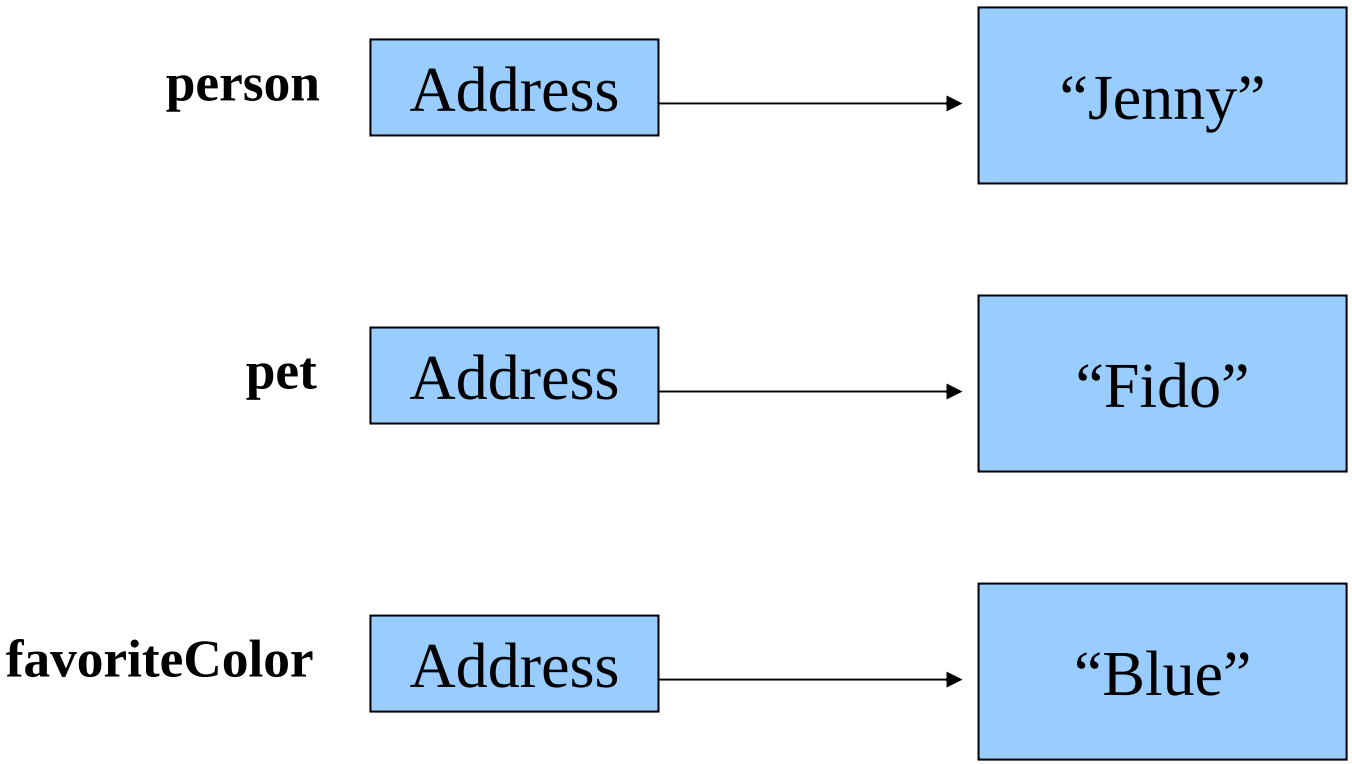
- Class objects normally have methods that perform useful operations on their data.

Classes and Instances

- Many objects can be created from a class.
- Each object is independent of the others.

```
string person = “Jenny”;
string pet = “Fido”;
string favoriteColor = “Blue”;
```

Classes and Instances



Classes and Instances

- Each instance of the class string contains different data.
- The instances are all share the same design.
- Each instance has all of the attributes and methods that were defined in the string class.
- Classes are defined to represent a single concept or service.

Building a `Rectangle` class

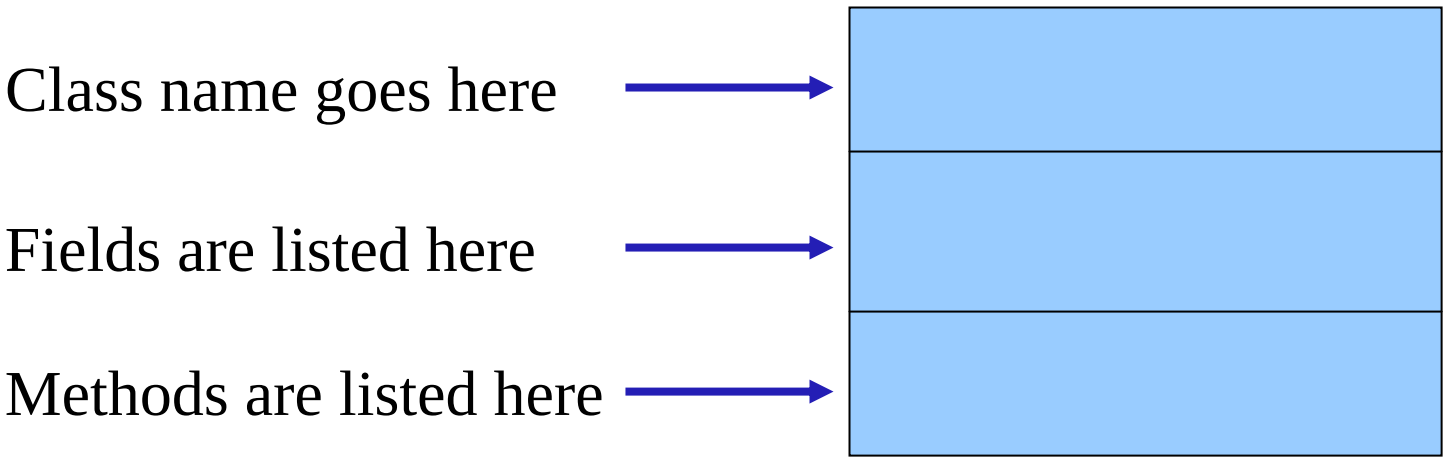
- A `Rectangle` object will have the following fields:
  - `length`. The length field will hold the rectangle's length.
  - `width`. The width field will hold the rectangle's width.

Building a `Rectangle` class

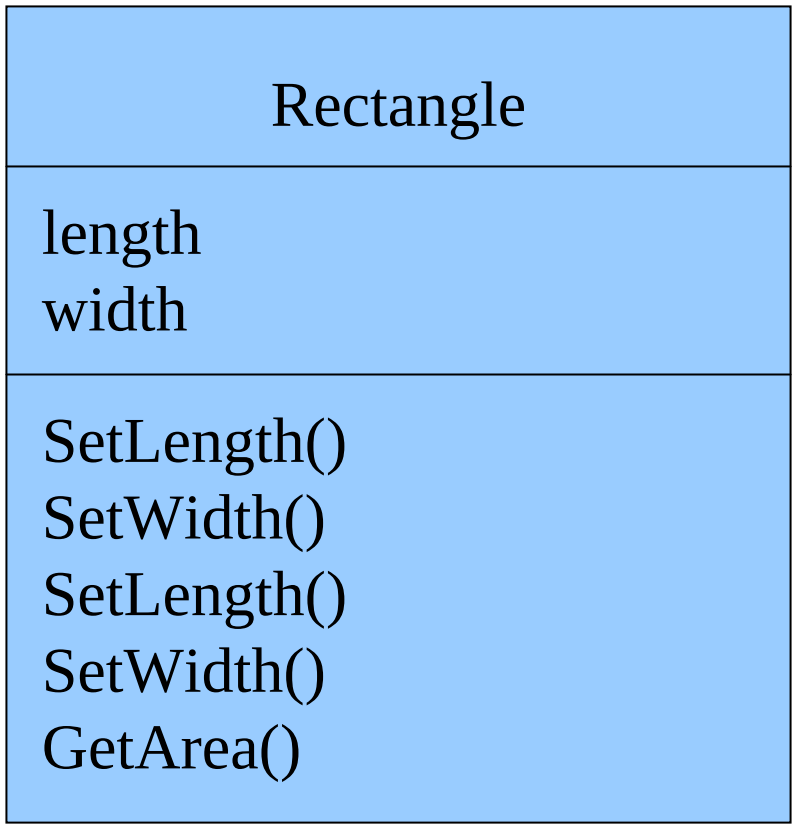
- The `Rectangle` class will also have the following methods:
  - `SetLength`. The `SetLength` method will store a value in an object's length field.
  - `SetWidth`. The `SetWidth` method will store a value in an object's width field.
  - `GetLength`. The `GetLength` method will return the value in an object's length field.
  - `GetWidth`. The `GetWidth` method will return the value in an object's width field.
  - `GetArea`. The `GetArea` method will return the area of the rectangle, which is the result of the object's length multiplied by its width.

UML Diagram

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting object-oriented systems.



UML Diagram for `Rectangle` class



Writing the Code for the Class Fields

```
public class Rectangle
{
    private double length;
    private double width;
}
```

Member visibility

- Fields (attributes) and methods of a class have also access rules.
- Accessibility modifiers `public, protected` and `private, internal` and `protected internal` can be used to set the visibility (accessibility) of the members (fields and methods) of a class.
- How a member is accessible depends on which access modifier they are declared with.

Access Specifiers

- An access specifier is a C# keyword that indicates how a field or method can be accessed.
- `public`
  - When the `public` access specifier is applied to a class member, the member can be accessed by code inside the class or outside.
- `private`
  - When the `private` access specifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.

Access Specifiers cont.

- `protected:`
  - Makes the member accessible by code in the same class as well as in its subclasses (this will be described in connection with inheritance i a later module).
- `internal:`
  - Makes the member accessible by code in the same assembly, but not outside the assembly.

Access specifiers cont.

- `internal protected`:
  - Works as internal but, it makes the member also accesible for sub-classes in other assemblies.
- Note:
  - A `namespace` does not have any access specifier.

Fields accessibility - style recommendation

- Fields of a class should always be declared as `private`.
- This is in accordance to one of the important principles of the object-oriented programming, namely encapsulation and data hiding.
- Only constants that are `static` can in some cases be declared as `public`.
- `public` fields should strictly be avoided unless there is a special reason for it.
- `public` fields are strictly forbidden in this course.

Methods accessibility- style recommendation

- Methods should be declared public when they provide services to other classes, i.e. when the method is a part of the interface of the class.
- Other methods should be declared as private, in the first place, and protected otherwise.
- Methods that are used only by other methods internally in the same class, should always be private.

Hide the internal methods

- When methods need to be called by other objects in a sequence, it may be practical to make them private and encapsulate them inside a public method
- This public method serves then as a part of the class's interface.

```
public void Calculate()
{
    ReadInput();     //private method
    Compute();       //private method
    ShowResults();   //private method
}
```

Encapsulation and data hiding

- A program in C# consists only of classes.
- A class is a collection of data and operations, or in programming terms, fields and methods.
- Object-oriented programming combines data and behavior via *encapsulation*.
- Encapsulation is a technique that seals the data and the internal methods safely inside the “capsule” of the class.
- Encapsulation is achieved effectively by hiding data, and declaring them as private members inside the class, where they can be accessed only by setter and getter methods (Properties in C# and VB).

Reasons for data hiding

- The most important reason is to hide the internal implementation details of the class.
- By doing so, you can safely modify the implementation without worrying about existing code that uses the class.
- Another reason for encapsulation is to protect your class against accidental or unwanted willful modifications.

Data hiding cont.

- A class often contains a number of interdependent fields that must be in a consistent state.
- By allowing a programmer (including yourself) to manipulate those fields directly, it can happen that changes may be done in one place and not all related fields may be updated, and thus putting the class in a inconsistent state.

Data hiding cont.

- To call a method to change the field, you gain a much better control to keep the state consistent.
- Classes can have certain methods and (even data) for internal use only. Hiding these methods prevents users of the class from calling them. Other classes may not even want to about these.
- Keeping the API of a class at a minimum makes the class easy to use and understand. If a field or method is visible to the users of your class, you have to document it. Save yourself time and effort by hiding it instead.

Classes accessibility

- How classes can be accessed (or visible) from outside the class, depends on how they are declared.
- A class in C# may be declared as `public`, `internal` or with no access modifier.
- The specifiers `private`, `protected` or `internal protected` cannot be used with class declarations.
- `public` classes are visible to all other classes everywhere, in all assemblies.
- `internal` classes are visible to all classes in the same namespace.

Class accesibility cont.

- When no access-modifier is specified, the class is internal by default.
- Classes usually need to be `public` so that other classes can make use of them.
- Unlike Java, more than one `public` class can be placed in the same file, it is very common to have one class per file, no matter how little the class may be.
- Unlike Java a class name does not need to match the file name.
- Unlike Java, the class structure, organized in namespaces and directories, need not to map to same file structure on the computer.

Properties

- Data hidden inside a class according to rules of OOP, need to be accessed by other classes.
- As a programmer, you can judge whether the data should be accessed as readonly, writeonly, both or not be accessed at all.
- The accessibility is done by methods that are commonly known as get and set methods, or getters and setters.

Setter and getter methods

- I most other programming languages, the methods are normal methods (as Set- and Get-methods in the Rectangle example given earlier), but in C#, these methods have been standardized.
- The methods are called Properties. An example is given in the next slide.
- A get-method is used to retrieve a value, while a set-method is used to change the value.
- Note that the word ”value” is keyword in C#.

Properties

- The keyword `value` has the same type as the return value for the property. If the return value is an object type, value will also be an object of the same type.

```
int count = 0;

//both read and write access
public int Count
{
    get{ return count;}
    set
    {
        if (count >= 0)
            count = value;
    }
}
```

Note: no argument

read access

write access

value is of type int here.

Properties

- Thre are two types of Property:
  - get – to read value of a field,
  - set – to change or update value of a field
- You can have one or both of these connected to a field.
- If you you only provide a get-property, the field becomes ”Readonly”.
- If you have only a set-property, the field becomes ”Writeonly”.
- The examples show all these forms.

Both get- and set properties

```
//Both read and write access
public int Count
{
    get{ return count;}
    set
    {
        if (count >= 0)
            count = value;
    }
}
```

Only get- or set property example

```
//Only read access
public string ID
{
    get { return id; }
}
//Only write access
public string Notes
{
    set { notes = value; }
}
```

value has the type string herec

Using properties

- Properties are used in the same way as instance fields.
  - objName.PropertyName
- Not that there is no parenthesis after the property name. Writing parentheses will cause a compiler error.
- The compiler knows which one of the get and set properties to call.

Using properties example

```
private void InitializeValues()
{
    Product product = new Product();
    product.Name = "Black Swan";  //set-property called here
    product.Number = 4;  //set-property called here

    //get-property Number called
    for (int i = 0; i < product.Number; i++)
    {
        //code
    }
}
```

get property automatically called here to retrieve value of Number

Constructors

- Classes can have special methods called *constructors*.
- A constructor is a method that is automatically called when an object is created.
- Constructors are used to perform operations at the time an object is created.
- Constructors typically initialize instance fields and perform other object initialization tasks.

Constructors

- Constructors have a few special properties that set them apart from normal methods.
- Constructors have the same name as the class.
- Constructors have no return type (not even `void`).
- Constructors may not return any values.
- Constructors are typically public.

Constructor for `Rectangle` Class

```
public class Rectangle
{
    private double length
    private double width;

    public Rectangle(double len, double wid)
    {
      length = len;
      width = wid;
    }
    //other methods
}
```

Uninitialized Local Reference Variables

- Reference variables can be declared without being initialized.

```
Rectangle box;
```

- This statement does not create a `Rectangle` object, so it is an uninitialized local reference variable.
- A local reference variable must reference an object before it can be used, otherwise a compiler error will occur.

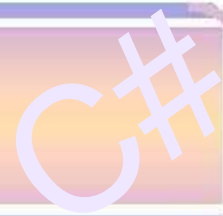
```
box = new Rectangle(7.0, 14.0);
```

- `box` will now reference a `Rectangle` object of length 7.0 and width 14.0.

The Default Constructor

- When an object is created, its constructor is always called.
- If you do not write a constructor, the C# compiler provides one when the class is compiled. The constructor that C# provides is known as the *default constructor*.
  - It sets all of the object's numeric fields to 0.
  - It sets all of the object's `bool` fields to `false`.
  - It sets all of the object's reference variables to the special value *null*.

The Default Constructor



- The default constructor is a constructor with no parameters, used to initialize an object in a default configuration.
- The only time that C# provides a default constructor is when you do not write any constructor for a class.
- A default constructor is not provided by C# if a constructor is already written.

Overloading Methods and Constructors

- Two or more methods in a class may have the same name as long as their parameter lists are different:
  - different types (or order of types),
  - different number of parameters.
- When this occurs, it is called *method overloading*. This also applies to constructors.
- Method overloading is important because sometimes you need several different ways to perform the same operation.

Overloaded Method add

```
public int add(int num1, int num2)
{
   int sum = num1 + num2;
   return sum;
}
public string add (string str1, string str2)
{
    string combined = str1 + str2;
   return combined;
}
```

Rectangle Class Constructor Overload

```
public Rectangle()
{
  length = 1.0;
  width = 1.0;
}
```

```
public Rectangle(double len,
                 double w)
{
  length = len;
  width = w;
}
```

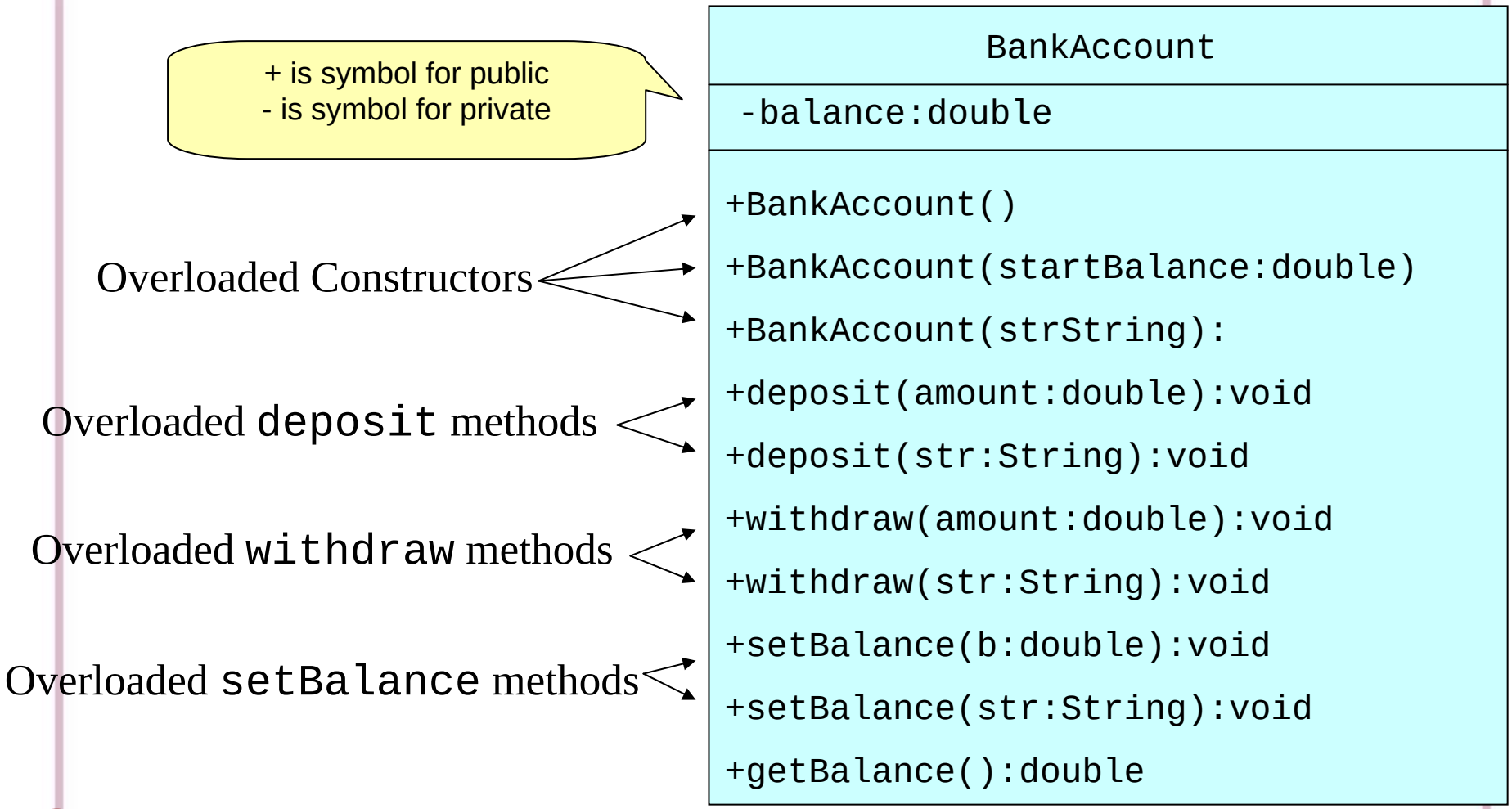
- If we were to add the default constructor we wrote previously to our Rectangle class in addition to the original constructor we wrote (as in above), what would happen when we execute the following calls?

```
Rectangle box1 = new Rectangle();
Rectangle box2 = new Rectangle(5.0, 10.0);
```

Answer to the previous question

- The first call would use the default constructor and box1 would have a length of 1.0 and width of 1.0.
- The second call would use the original constructor and box2 would have a length of 5.0 and a width of 10.0.

The BankAccount Example



Summary

- Begin class names with a capital letter.
- Instance variables can be of primitive types or of reference types, i.e. other classes.
- Each object of the class is independent of each other and has a separate instance of the variables.
- All class fields should be private. This is *mandatory* in this course!
- Declare only those methods that form a part of the interface of the class as public.