

Object-Oriented Programming Using Microsoft Visual C# .NET

Using Classes and Objects

Classes in OOP

- Classes model real-world objects and define
 - Attributes (state, properties, fields)
 - Behavior (methods, operations)
- Classes describe structure of objects
 - Objects describe particular instance of a class
- Properties hold information about the modeled object relevant to the problem
- Operations implement object behavior

Object-Oriented Programming

Fundamental Principles

- Encapsulation

It is also called "**information hiding**". An object has to provide its users only with the essential information for manipulation, without the internal details.

- Inheritance

Inheritance is a fundamental principle of object-oriented programming. It allows a class to "inherit" (behavior or characteristics) of another, more general class.

- Abstraction

Abstraction means working with something we know how to use without knowing how it works internally. To deal with objects considering their important characteristics and ignore all other details.

- Polymorphism

To work in the same manner with different objects, which define a specific implementation of some **abstract behavior**.

Classes in C#

- Classes in C# could have following members:
 - Fields, constants, methods, properties, indexers, events, operators, constructors, destructors
 - Inner types (inner classes, structures, interfaces, delegates, ...)
- Members can have access modifiers (scope)
 - `public`, `private`, `protected`, `internal`
- Members can be
 - `static` (common) or specific for a given object

Simple Class Definition

```
public class Cat  
{
```

Begin of class definition

```
    private string name;  
    private string owner;
```

Fields

```
    public Cat(string name, string owner)
```

```
    {  
        this.name = name;  
        this.owner = owner;  
    }
```

Constructor

```
    public string Name  
    {  
        get { return name; }  
        set { name = value; }  
    }
```

Property

Simple Class Definition (2)

```
public string Owner
{
    get { return owner;}
    set { owner = value; }
}
```

Method

```
public void SayMiau()
{
    Console.WriteLine("Miauuuuuuu!");
}
}
```

**End of class
definition**

Class Definition and Members

- Class definition consists of:
 - Class declaration
 - Inherited class or implemented interfaces
 - Fields (static or not)
 - Constructors (static or not)
 - Properties (static or not)
 - Methods (static or not)
 - Events, inner types, etc.

Access Modifiers

Public, Private, Protected, Internal

Access Modifiers

- Class members can have access modifiers
 - Used to restrict the classes able to access them
 - Supports the OOP principle "encapsulation"
- Class members can be:
 - `public` – accessible from any class
 - `protected` – accessible from the class itself and all its descendent classes
 - `private` – accessible from the class itself only
 - `internal` – accessible from the current assembly (used by default)

Using Classes and Objects

Using Classes

- How to use classes?
 - Create a new instance
 - Access the properties of the class
 - Invoke methods
 - Handle events
- How to define classes?
 - Create new class and define its members
 - Create new class using some other as base class

How to Use Classes (Non-static)?

1. Create an instance
 - Initialize fields
2. Manipulate instance
 - Read / change properties
 - Invoke methods
 - Handle events
3. Release occupied resources
 - Done automatically in most cases

Constructors

Defining and Using Class Constructors

What is Constructor?

- Constructors are special methods
 - Invoked when creating a new instance of an object
 - Used to initialize the fields of the instance
- Constructors has the same name as the class
 - Have no return type
 - Can have parameters
 - Can be `private`, `protected`, `internal`, `public`

Defining Constructors

- ◆ Class `Point` with parameterless constructor:

```
public class Point
{
    private int xCoord;
    private int yCoord;

    // Simple default constructor
    public Point()
    {
        xCoord = 0;
        yCoord = 0;
    }

    // More code ...
}
```

Defining Constructors (2)

```
public class Person
{
    private string name;
    private int age;

    // Default constructor
    public Person()
    {
        name = null;
        age = 0;
    }

    // Constructor with parameters
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // More code ...
}
```

As rule constructors should initialize all own class fields.

Properties

Defining and Using Properties

The Role of Properties

- Expose object's data to the outside world
- Control how the data is manipulated
- Properties can be:
 - Read-only
 - Write-only
 - Read and write
- Give good level of abstraction
- Make writing code easier

Defining Properties

- Properties should have:
 - Access modifier (`public`, `protected`, etc.)
 - Return type
 - Unique name
 - `Get` and / or `Set` part
 - Can contain code processing data in specific way

Defining Properties – Example

```
public class Point
{
    private int xCoord;
    private int yCoord;

    public int XCoord
    {
        get { return xCoord; }
        set { xCoord = value; }
    }

    public int YCoord
    {
        get { return yCoord; }
        set { yCoord = value; }
    }

    // More code ...
}
```

Dynamic Properties

- Properties are not obligatory bound to a class field
 - can be calculated dynamically

```
public class Rectangle
{
    private float width;
    private float height;

    // More code ...

    public float Area
    {
        get
        {
            return width * height;
        }
    }
}
```

Automatically Implemented Properties

- Properties could be defined without an underlying field behind them
 - It is automatically created by the compiler

```
class UserProfile
{
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
...
UserProfile profile = new UserProfile() {
    FirstName = "Steve",
    LastName = "Balmer",
    UserId = 91112 };
```

Static Members

Static vs. Instance Members

Static Members

- Static members are associated with a type rather than with an instance
 - Defined with the modifier `static`
- Static can be used for
 - Fields
 - Properties
 - Methods
 - Events
 - Constructors

Static vs. Non-Static

- **Static:**
 - Associated with a type, not with an instance
- **Non-Static:**
 - The opposite, associated with an instance
- **Static:**
 - Initialized just before the type is used for the first time
- **Non-Static:**
 - Initialized when the constructor is called

COLLECTIONS

Lists

- The most common sort of collection is a `List<T>`. Once you create a `List<T>` object, it's easy to add an item, remove an item from any location in the list, peek at an item, and even move an item from one place in the list to another.
- Here's how a list works:
 - First you create a new instance of `List<T>`
 - You need to specify the type of object or value that the list will hold by putting it in angle brackets `<>` when you use the `new` keyword to create it.
 - `List<Card> cards = new List<Card>();`
- The `<T>` at the end of `List<T>` means it's *generic*.
- The `T` gets replaced with a type—so `List<int>` just means a List of ints.

Lists




- A List **resizes dynamically** to whatever size is needed.
- To put something into a List, use **Add()**.
- To remove something from a List, use **Remove()**.
- You can remove objects using their **index** number using **RemoveAt()**.
- To find out where something is (and if it is) in a List, use **IndexOf()**.
- To get the number of elements in a List, use the **Count** property.
- You can use the **Contains()** method to find out if a particular object is in a List.

Lists

- The `List` collection has some important properties and methods.

```
private List <Flat> flats= new List<Flat>();  
.  
.  
flat = new Flat(idcode, addr, area,  
    NumOfRooms, comfortLevel);  
  
flats.Add(flat);
```

Properties

	Name
	Capacity
	Count
	Item

Files

- Contain a sequence of bytes that represent information
- Persisting objects to files is important because
 - It eliminates need to reenter data
 - It allows information to be shared between people, or applications
- File format
 - Describes how file is organized
 - Determined by software used to create the file

Files (continued)

- File extension
 - Does not dictate what is contained within the file
- To identify a file's location, you need
 - The drive letter of the disk
 - The folders and subfolders in which the file is contained
 - The filename
 - The file extension

The .NET System.IO Namespace

- Classes can be grouped into two functions
 - Classes that manipulate the file system
 - Classes for saving and retrieving data to and from external storage devices

System.IO.Directory

- Directory class
 - Exposes methods that can be used to create, delete, move, and list subfolders and files
- Because methods are defined as *static*
 - They must be called using class name and not using an instance variable name

System.IO.File

- File class
 - Exposes methods that can be used to copy, delete, and move files
- File methods
 - Defined as *static*
 - Must be called using the class name and not using an instance variable name

System.IO.Path

- Methods in the Path class
 - Do not actually manipulate folders and files
 - Can be used to manipulate the strings that define a folder's or file's path

Types of Files

- Data
 - Can be stored in a file as plain-text strings or as binary data
- Text files
 - Usually have a file extension of .txt
- Saving data to a binary data file
 - Executes more efficiently than saving data to a text file

Persistence Using Sequential Text Files

- .NET *StreamReader* and *StreamWriter* classes
 - Used to read and write sequential text files with C#
- Hard-coded paths
 - Not a good idea because a path that exists on your computer might not exist on another computer
 - Can result in exceptions

Fixed-Width Text Files

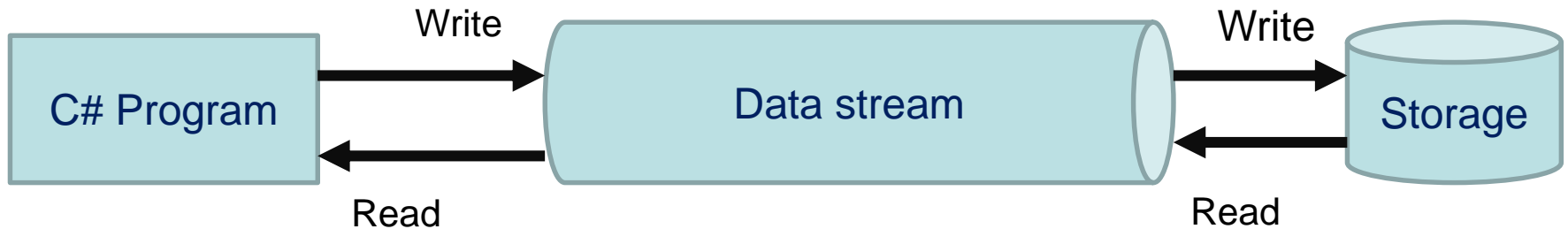
- Format
 - File contains zero or many records
 - Each record contains information about a single entity
 - Each record normally is terminated by a CRLF
- Contents of each field in each record is always the same number of characters
- Nothing in the file delimits where a field begins and ends

Fixed-Width Text Files (continued)

- Processing
 - First open the file
 - For reading or writing records
 - Use the *ReadLine* method of the *StreamReader* class or the *WriteLine* method of the *StreamWriter* class
 - File is closed when you are finished using it
- Parsing input records and formatting output records
 - Requires more tedious coding than with tab-delimited records

Using Files

- All input and output in the .NET Framework involves the use of streams. A stream is an abstract representation of a serial device. (network channel, disk, memory location etc.) A data stream is the flow of data from a source to a single receiver.

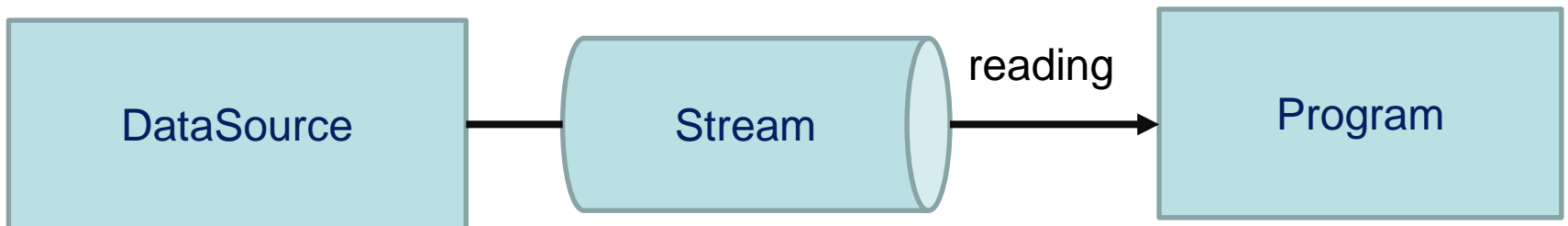


Whenever you want to read data from a file or write data to a file, you'll use a Stream object.

Reading – Writing

Reading

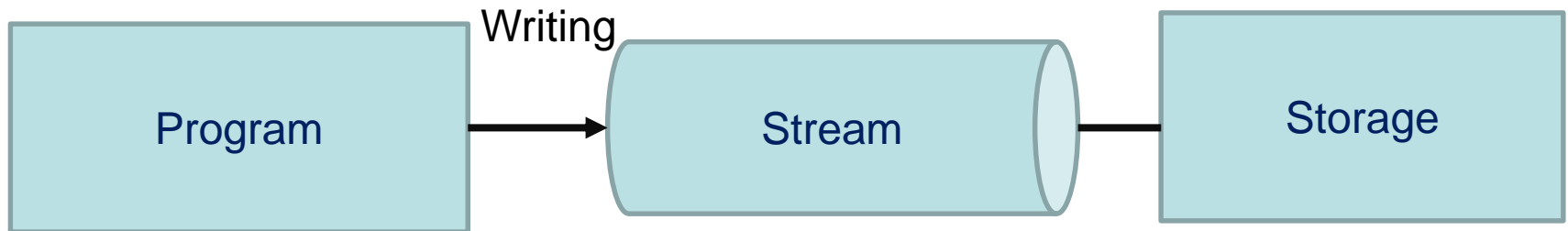
- Open the stream
- while there is new data
 - Read
- Close the stream



Reading – Writing

Writing

- Open the stream
- while there is data
 - write
- close the stream



Example

```
// idNumber is the serial number of reading
int idNumber = 1;

StreamReader streamReader = new StreamReader("competitors.txt");

while (!streamReader.EndOfStream) {
    name = streamReader.ReadLine();
    departement = streamReader.ReadLine();

    competitor = new Competitor(idNumber, name, departement);
    // The competitor is added to the list of competitors
    competitors.Add(competitor);

    idNumber++;
}
streamReader.Close();
```