Strings

sometime behave a bit devilish

- Variables of type `string`
- they are objects but can be directly initialized:
- `string s="piece of text";`
- Comparison at equality compares the string-content
- (however, in family of C language it is exceptional).
- Length – as in arrays: `s.Length`
- String can be considered as an array,
- but it is read-only! For writing use `StringBuilder`.
- Array can be splitted into array using its method `Split`:
- ```
  string t="1 22 333 444";
  string[]nums=t.Split({' '});
  ```
  ```

Structures

- In C programming language there were no objects but there were structures.
- They were operated as records in Pascal,
- they were defined using keyword `struct` (syntactically similarly to classes in C#),
- In C# they are present too, but instead of them one uses usually objects.
- Example: `struct compl{public int re,im;};`

Inheritance

- Sometimes we have special cases (to the general case) and we want them to behave uniformly.
- E.g., a living mass has 7 methods (signs of life), but each living mass behaves completely differently.
- If we want to implement menagerie, it would be nice to implement a type `animal` that would be kind of template for particular animal species.
- Defining a "parent" is simple, son gets defined using semicolon-operator.

Example

parent

```
class animal {
      string name;
      public void MakeNoise()
      { Console.WriteLine("Cannot, I am animal!");}
      public void SetName(string name)
      { this.name=name;}
      public void WhoSThat()
      { Console.WriteLine(name);}
}
```

Example

sons

```
class tiger:animal
{    public tiger(string name)
     {    SetName(name);}
     public void MakeNoise()
     {    Console.Writeline("Vrrrrrr-rrum!");}
}
class hen:animal
{   public hen() //hens have no names
    { SetName("none"); }
    public void MakeNoise()
    {    Console.WriteLine("Ko - ko - ko!");}
}
```

Inheritance

and its use

- The son-class inherites everything from the parent class,
- if we redefine some method, it is redefined,
- however, we would appreciate some "uniform approach"to the sons (otherwise inheritance is just a toy), thus
- we may assign a son into a parent
- `animal matysek=new tiger("Matysek");`
- Then we can call inherited methods: `matysek.WhoSThat();`
- We may also try to call redefined methods (defined in parent),
- but it does something unexpected: `matysek.MakeNoise();` `=> "Cannot, I am animal!''`
- To make possible a behavior we expect, the method would have to be virtual.

Virtual methods I/II

motivation by example

- If the previous problem was unsolvable, object programming would be merely useless.
- Thus it can be solved using **virtual methods.**
- Defining virtual methods – modifier `virtual`.
- Example: `public virtual void MakeNoise()...` and `public override MakeNoise()...`
- After this modification the example does what we wanted.

Virtual methods II/II

explanation how it works

- Although methods behave like if they were present in each individual object (they can operate its attributes), it would be too demanding...
- ...thus they are stored in a class-prototype.
- Thus methods redefined in son-class are not reflected when referencing the object as parent
  `animal a=new tiger();`
- Even virtual methods are not present in each object, they are represented by VMT and VMT is also present in prototype.
- But each object has a pointer at VMT and this pointer gets initialized when calling the constructor.
- That is why it works as we want.

Abstract classes I/II

and abstract methods

- Sometimes we want to use classes only as some template – we define what each class inheriting from us must define without defining general version.
- Example: Printer has method `print`, but ink-printer and laser-printer are doing it in completely different way.
- Abstract class and abstract (purely virtual) methods are giving us this possibility.
- Abstract method is not defined, just declared. In C# it is denoted by modifier `abstract`
- `public abstract void print(string x);`

Abstract classes II/II

- Abstract class is a class containing at least one abstract method.
- In C# it must be also denoted by keyword `abstract`
- ```
  abstract class printer{
  public abstract void print(string x);}
  ```
- Abstract classes cannot be instantiated, they just define what each (non-abstract) descendant must define.
  ```

Sealed classes and methods

- Sealed classes cannot be used as parents of further classes,
- sealed methods we cannot override.
- Modifier `sealed` is used in a (syntactically) similar way as `abstract`.
- `sealed class thelastone{...}`,
- `public sealed override void print(){...}`.

Wrappers around attributes

when we want to be politically correct

- We know that attributes should not be visible,
- but we want to modify them often and calling functions is impractical.
- Thus we may establish entity resembling a variable consisting (in fact) of two methods (`get` and `set`).
- ```
  public int wrappername{
      get{   return Variable;}
      set{   Variable=value;}
  }
  ```
- `set` has an implicit argument `value`,
- `wrappername` then behaves as a variable.
  ```

Wrapper example

```
class animal
{    private int numLegs;
     public int NumLegs
     {    get
          {    return numLegs;}
          set
          { if((value%2)==0) numLegs=value;}
     }
}
```

Example – use

of wrapper around attribute

```
matysek=new tiger("Matysek");
matysek.NumLegs=4; //OK
matysek.NumLegs=3; //K. O.
Console.WriteLine(matysek.NumLegs);
```