Lecture 2
.NET Data/Memory Model

Programming II

School of Business Informatics
Spring 2018

(: 6 stages of debugging:

1. That can't happen 2. That doesn't happen on my computer
3. That shouldn't happen 4. Why does that happen?
5. Oh, I see. 6. How did that ever work? :)

Key points from last week

- Classes serve as blueprints for objects (class instances)
- Class data is stored in fields, which should not be accessible from outer classes
- To access hidden fields either methods or properties can be used

Auto properties

Auto properties enable a much shorter definition than full-size properties. They are used for scenarios when getter and setter blocks have just a single assignment or return statement:

```
class Student
{
   public string Name { get; set; } // This line declares both
       a field and a getter,setter around it
}
```

Auto property pitfall

When declaring auto-properties don't declare an additional field!

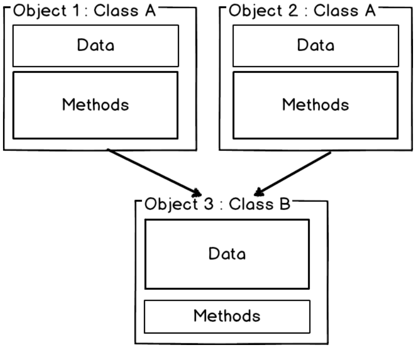
```
class Student
{
   string _name;     // Excessive

   public string Name { get; set; } // 3 in 1: field, getter,
       setter
}
```

Object initialization

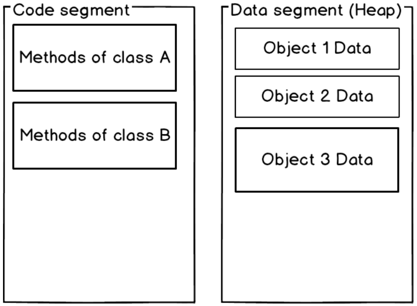
- A **constructor** is a special method inside a class that is responsible for initialization of an object's state
- A constructor is called only **once** when a new object is created
- A class can have more than one constructor. In this case all class constructors have to differ in signature
- In case no constructors are defined in a class, the compiler automatically inserts a default constructor
- Constructors can invoke each other - see example

Structure of an OO program

Logically a OO program is formed by a number of objects, each containing data and methods for working with it



Physically methods and data are placed in different areas of memory (code segment and data segment respectively)



Classes in C#

C# classes can contain:

- Fields
- Methods
- Properties
- Constructors
- Events

Classes in C#

C# classes can contain:

- Fields
- Methods
- Properties
- Constructors
- Events

Which of the items above relate to data and which to code?

Classes in C#

C# classes can contain:

- Fields - data
- Methods - code
- Properties - code
- Constructors - code
- Events - data

Properties or methods?

Both properties and methods can be used to access private data inside the class.

- In general, properties represent data while methods represent actions
- Properties should not contain complicated calculations
- A class needs to be designed in a way that its properties can be set in any order

More advice on how to use properties and methods on MSDN

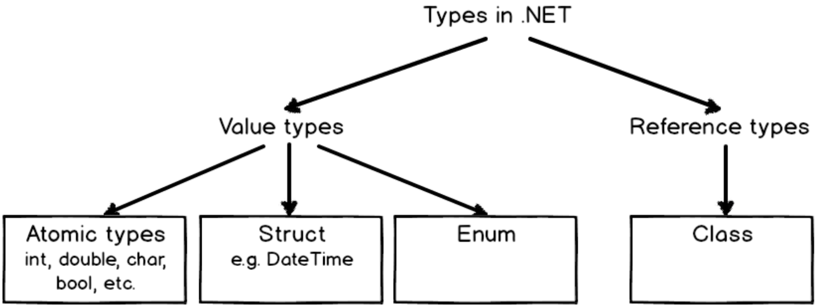
Data model of .NET programming language

The data (or memory) model of .NET is based on the following three mechanisms:

- Value and reference types
- Heap and stack
- Mutable and immutable objects

These three mechanisms should be viewed independently even though they have certain connections with each other.

.NET types


Types in .NET
Value types
Reference types
Atomic types
int, double, char, bool, etc.
Struct
e.g. DateTime
Enum
Class

Values and references

- Variables of value types store data in the place where they are declared
- Reference variables store address of a block of memory where data resides
- A reference variable has a value of **null** when it does not point to any allocated block of memory
- When assigning variables of value types a new independent copy of the original value is made. When assigning variables of reference types, only references are copied (object data is shared among several references)

Heap and stack

.NET applications use two types of memory: a heap and a stack.

The **stack** is dynamically changed as the program enters and leaves methods. It stores:

- Local variables declared inside methods
- Method parameters
- Return addresses forming a chain of method calls

The **heap** is a much larger block. It stores data of all objects (instances of classes)

Class or struct

- Both classes and structs in C#:
    - Are containers for structuring data
    - Group together data and related code
    - Can contain fields, methods, properties and constructors
- Structures do not support inheritance
- Structures are value types whereas classes are reference types

Most types in .NET are classes.

Immutable objects

An immutable object cannot change its state (contents) once created.
Instead a new object is instantiated each time a source object is changed.

Immutable objects

An immutable object cannot change its state (contents) once created. Instead a new object is instantiated each time a source object is changed.

- Immutability has nothing to do with value or reference types. It results from the way a class is designed
- There is however a strong recommendation to always make a struct immutable as the opposite may lead to errors (see MutableStruct project in the supplement)

Immutable objects

An immutable object cannot change its state (contents) once created. Instead a new object is instantiated each time a source object is changed.

- Immutability has nothing to do with value or reference types. It results from the way a class is designed
- There is however a strong recommendation to always make a struct immutable as the opposite may lead to errors (see MutableStruct project in the supplement)

Examples of standard immutable types: DateTime, string

Immutability guideline I

When calling a method that is aimed at changing an immutable object don't forget to assign the result, otherwise the change will be immediately lost:

```
string city = "london";
city.ToUpper();           // No change!!!
city = city.ToUpper();    // Correct form
```

Immutability guideline II

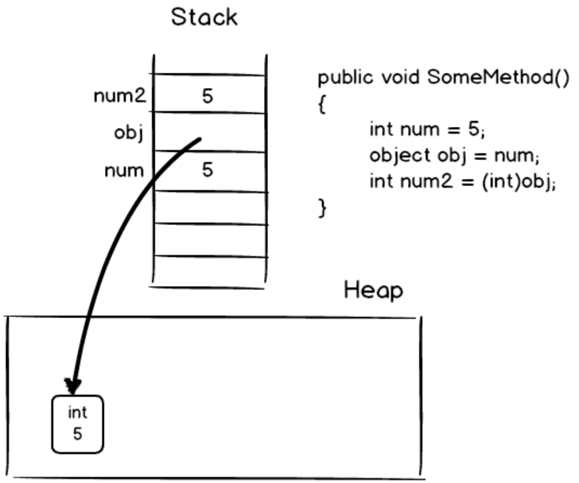
When constructing a string of a large number of parts, use the StringBuilder class rather than string concatenation. The latter produces too many temporary string objects.

```
static string ArrayToStringInefficient(int[] array) {
    string result = "";
    for (int i = 0; i < array.Length; i++) {
        result += array[i].ToString() + " ";
    }
    return result;
}

static string ArrayToStringEfficient(int[] array) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < array.Length; i++) {
        sb.Append(array[i]);
        sb.Append(' ');
    }
    return sb.ToString();
}
```

Boxing and unboxing

- Any .NET type can be converted to object (System.Object)
- If the converted type is a value type, boxing occurs.
- When converting from a reference type to a value type the variable is unboxed.
- Often explicit boxing happens, e.g. when passing a value type to a method accepting object.

Boxing/unboxing example

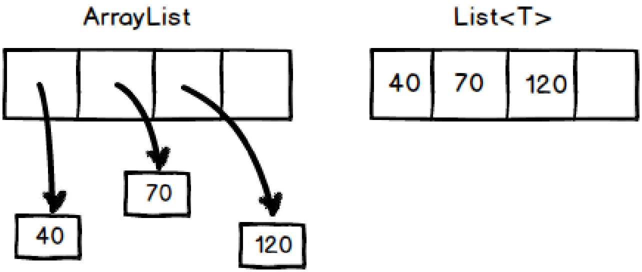


```
public void SomeMethod()
{
        int num = 5;
        object obj = num;
        int num2 = (int)obj;
}
```

Boxing costs: demo

“Boxing” project in the supplement:

- An Arraylist internally stores data as an array of objects. Each value type variable is boxed
- A List<T> stores values inside a single container



Static vs non-static

Classes can have both static and non-static members. Static members are declared with an additional **static** modifier

Static members belong to the class (shared among all objects) while non-static members relate to individual objects

Static and non-static call rules

1. A non-static member (method, property) can access both static and non-static members of the same class
2. A static member can directly access only static members of the same class