Object-oriented programming #7-1
Type converters, structures

Wojciech Complak

Institute of Computing Science
Faculty of Computing
Poznan University of Technology

e-mail: Wojciech.Complak@wsb.poznan.pl

1

Lecture content

- type converters
- structures

2

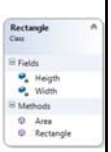
Converters (#1/6)

defining type converters

```
class Square
{
  uint Side;
  public Square(uint newSide) { this.Side = newSide; }
  public double Area() { return this.Side * this.Side; }
}
```

task: define explicit type converter from class **Square** objects to class **Rectangle** objects

```
class Rectangle
{
  uint Width, Heigth;
  public Rectangle(uint newHeight, uint newWidth)
  {
    this.Heigth = newHeight;
    this.Width = newWidth;
  }
  public double Area() { return this.Heigth * this.Width; }
}
```

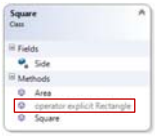


3

Converters (#2/6)

defining *explicit* type converter

```
class Square
{
  /* ... */
  public static explicit operator Rectangle(Square s)
  {
    return new Rectangle(s.Side, s.Side);
  }
}
```



explicit type converter definition: **Square -> Rectangle**

```
class Rectangle
{
  uint Width, Heigth;
  public Rectangle(uint newHeight, uint newWidth)
  { /* ... */ }
  /* ... */
}
```

```
Square s = new Square(5);
Rectangle r = (Rectangle)s;
```

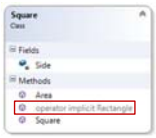
using the explicit type converter **Square -> Rectangle**

4

Converters (#3/6)

defining *implicit* type converter

```
class Square
{
  /* ... */
  public static implicit operator Rectangle(Square s)
  {
    return new Rectangle(s.Side, s.Side);
  }
}
```



implicit type converter definition: **Square -> Rectangle**

```
class Rectangle
{
  uint Width, Heigth;
  public Rectangle(uint newHeight, uint newWidth)
  { /* ... */ }
  /* ... */
}
```

```
Square s = new Square(5);
Rectangle r1 = (Rectangle)s;
```

explicit conversion **Square -> Rectangle**

```
Rectangle r2 = s;
```

implicite conversion **Square -> Rectangle**

5

Converters (#4/6)

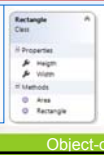
bidirectional converters

```
class Square
{
  public uint Side { private set; get; }
  public Square(uint newSide)
  { this.Side = newSide; }
  public double Area()
  { return this.Side * this.Side; }
}
```

- converters can be added in any combination to both classes (without repetitions of the signature),
- converters must have access to the necessary data from the source class (encapsulation)

```
public static implicit operator Square(Rectangle s)
{
  return new Square((s.Width + s.Heigth) / 2);
}
public static implicit operator Rectangle(Square s)
{
  return new Rectangle(s.Side, s.Side);
}
```

```
class Rectangle
{
  public uint Width { private set; get; }
  public uint Heigth { private set; get; }
  public Rectangle(uint newHeight, uint newWidth)
  {
    this.Heigth = newHeight;
    this.Width = newWidth;
  }
  public double Area()
  { return this.Heigth * this.Width; }
}
```

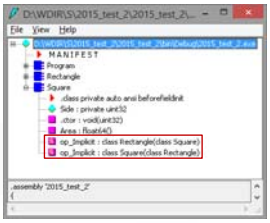


6

Object-oriented programming

Converters – remarks (#5/6)

- *explicit* type converter requires using casting (),
- *implicit* type converter supports conversion with or without casting,
- for the same combination of input and output types one cannot define both *implicit* and *explicit* type converter
- implicit converter ☞ if conversion cannot cause data (precision) loss (e.g. *byte* -> *uint*),
- converters are implemented as overloaded static methods *op_Implicit()* and *op_Explicit()*



Object-oriented programming (7/32)

7

Object-oriented programming

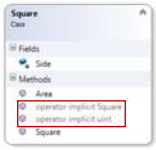
Converters (#6/6)

converters can also be created for built-in/new type

```
class Square
{
  /* ... */
  public static implicit operator uint (Square s)
  {
    return s.Side;
  }
  public static implicit operator Square (uint s)
  {
    return new Square(s);
  }
}
```

implicit type converter: `Square -> uint`

implicit type converter: `uint -> Square`



Object-oriented programming (8/32)

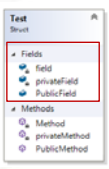
8

Object-oriented programming

Structures: instance fields (#1/23)

```
struct Test
{
  public int PublicField /* = 3 */;
  private int privateField /* = 5 */;
  /* private */ int field;
  public void PublicMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
  private void privateMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.PublicMethod();
  }
  /* private */ void Method() { }
}
```

- fields are by default *private*: they are accessible for the structure methods, but inaccessible from outside the structure,
- public fields (accessible both for structure methods and from outside of the structure) must be explicitly flagged as *public*,
- structure fields cannot be *protected*: structures do not inherit from classes/structures and are by definition final (*sealed*)
- fields cannot be initialised in the structure declaration



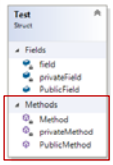
Object-oriented programming (9/32)

9

Object-oriented programming

Structures: instance methods (#2/23)

```
struct Test
{
  public int PublicField /* = 3 */;
  private int privateField /* = 5 */;
  /* private */ int field;
  public void PublicMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
  private void privateMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.PublicMethod();
  }
  /* private */ void Method() { }
}
```



- methods are *private* by default: they are accessible for the structure methods, but inaccessible from outside the structure,
- *public* methods (available both for structure methods and from outside the class) must be explicitly flagged as *public*,
- structure methods cannot be *protected*: structures do not inherit from classes/structures and are by definition final (*sealed*)
- methods have access to all members of the structure
- the *this* keyword lets you refer to members of the current structure instance

Object-oriented programming (10/32)

10

Object-oriented programming

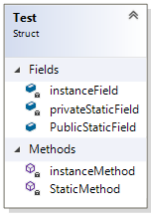
Structures: static members (#3/23)

- static members can be used even when no instance of the structure exists,
- static field declaration creates one instance of a variable shared by all instances of the structure,
- static fields are created and initialised during program start-up

```
struct Test
{
  public static int PublicStaticField = 3;
  private static int privateStaticField = 5;
  private int instanceField;
  static void StaticMethod()
  {
    PublicStaticField++;
    privateStaticField++;
    // this.instanceField++;
    // this.instanceMethod();
  }
  private void instanceMethod()
  {
    PublicStaticField++;
    privateStaticField++;
    this.instanceField++;
    StaticMethod();
  }
}
```

- static fields can be initialised in the structure declaration

- static methods can access only static members (fields and methods), they cannot access instance components,
- instance methods can access both static members and instance (non-static) members



Object-oriented programming (11/32)

11

Object-oriented programming

Structures: accessing members (#4/23)

```
struct Test
{
  public static void StaticMethod() { }
  public void InstanceMethod () { }
}
// ...
Test.StaticMethod();

Test t1;
t1.InstanceMethod();
```

access to static components using the type name - regardless of whether there is any instance of the structure

access to non-static components using instance (variable name instead of a reference) – a structure instance must exist

C# does not support static structures (they would be redundant in the language - identical to static classes)

Object-oriented programming (12/32)

12

Object-oriented programming

Structures: constructor(s) (#5/23)

- an auto-implemented argumentless default constructor is automatically generated for a structure (sets fields to default values)
- custom default constructor cannot be defined
- defining own constructor does not delete the auto-implemented default constructor

```
struct Test
{
  public double X, Y;
}
// ...
Test t1,
     t2 = default(Test),
     t3 = new Test();

t2.X = t2.Y = 3; // field modification

t2 = new Test();
```

a structure is created but the default constructor will not be run: structure fields are not initialised (for a class, this declaration would be a *null* reference)

invoking the default constructor

calling the ctor does not create a new object, instead it reinitialises the existing one

Object-oriented programming (13/32)

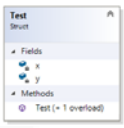
13

Object-oriented programming

Structures: constructor(s) (#6/23)

- constructor may be public or private (accessible only for structure methods)
- structure constructors may be interlinked just as between class constructors

```
struct Test
{
  private double x, y;
  private Test(double newX, double newY)
  {
    this.x = newX;
    this.y = newY;
  }
  public Test(string newX, string newY) :
         this(double.Parse(newX),  double.Parse(newY)) {  }
}
// ...
Test t1,
     t2 = default(Test),
     t3 = new Test(),
     // t4 = new Test(1.0, 2.0),
     t5 = new Test("1.0", "2.0");
```



no constructor will be run

default constructor will be called

default constructor will be called

the only compatible constructor is not accessible (*private*)

public constructor will be called

Object-oriented programming (14/32)

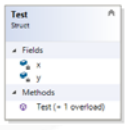
14

Object-oriented programming

Structures: copy constructor (#7/23)

shallow copy constructor is hardly useful: it only copies fields from the source

```
struct Test
{
  private double x, y;
  public Test(double newX, double newY)
  {
    this.x = newX;
    this.y = newY;
  }
  public Test(Test src)
  {
    this = src;
  }
}
// ...
Test t1 = new Test(1, 2),

     t2 = new Test(t1);
```



equivalent to:
`t2 = t1;`

Object-oriented programming (15/32)

15

Object-oriented programming

Structures: copy constructor (#8/23)

but the deep copy constructor (struct has at least one reference field) ...

```
struct MyVector
{
 private double[] vector;
 public MyVector (int newSize, double defaultValue = 0.0)
 {
   vector = new double[newSize];
   for (int i = 0; i < this.vector.Length; ++i)this.vector[i] = defaultValue;
 }
 public MyVector(params double[] newValues)
 {
   vector = new double[newValues.Length];
   for (int i = 0; i < this.vector.Length; ++i)this.vector[i] = newValues[i];
 }
 public MyVector(MyVector src)
 {
   this.vector=(double[])src.vector.Clone();
 }
}
```

this constructor is not equivalent to a shallow copy:

```
MyVector t1 = new MyVector (1, 2),
         t2;
t2=t1;                    // shallow copy
t2=new MyVector(t1); // deep copy
```

Object-oriented programming (16/32)

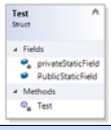
16

Object-oriented programming

Structures: static constructor (#9/23)

static constructor is used to initialise static fields

```
struct Test
{
  public static int PublicStaticField;
  private static int privateStaticField;
  static Test()
  {
    PublicStaticField = 3;
    privateStaticField = 5;
  }
}
```



static constructor:
- is run once during program start-up,
- can access only static members,
- cannot have any access modifiers,
- must be argumentless,
- is not directly runnable

Object-oriented programming (17/32)

17

Object-oriented programming

Structures: constants and *readonly* fields (#10/23)

constants and readonly fields

```
struct Test
{
  public const int Field1 = 2;

  private readonly int field2 /*= 2*/;
  private Test(int newField2)
  {
    field2 = newField2;
  }

  private static readonly int field3 = 3;
  static Test()
  {
    field3 = 3;
  }
}
```

- constants (*const*) can be structure fields (usually) or local in the member method,
- constant fields can be *private* (default) or *public*

readonly instance field can be initialised only within a constructor (in classes one can also use assignment)

static readonly field can be initialised both int the static constructor or with an assignment

Object-oriented programming (18/32)

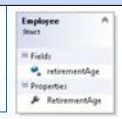
18

Structures: properties (#11/23)

structures may contain properties

```
struct Employee
{
  private uint retirementAge /*= 0*/;
  public uint RetirementAge
  {
    get { return retirementAge; }
    set { retirementAge = value; }
  }
}
```

backing field must be initialised in a constructor



```
struct Employee
{
  public uint RetirementAge
  {
    get;
    set;
  }
}
```

auto-implemented property must be initialised in a constructor

19

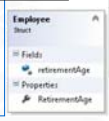
Structures: properties (#12/23)

properties can be:
- read-only– no *set* mutator
- write-only (excluding auto-implemented) – no *get* accessor

properties/accessors/mutators are by default *private*, to make them accessible from outside the structure access modifier *public* must be explicitly used

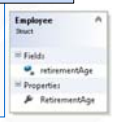
```
struct Employee
{
  private uint retirementAge;
  public uint RetirementAge
  {
    get { return retirementAge; }
  }
}
```

read-only property



```
struct Employee
{
  private uint retirementAge;
  public uint RetirementAge
  {
    set { retirementAge = value; }
  }
}
```

write-only property



20

Structures: *ToString()* method (#13/23)

overriding *ToString()* method inherited from *Object* type

```
struct My2dPoint
{
  private int x, y;
  public My2dPoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
  public override string ToString()
  {
    return base.ToString() + " x:" + this.x + " y:" + this.y;
  }
}
```

structures (implicitly) inherit from *System.Object* and *System.ValueType*

if we're interested in the name of type (works for structures too)

```
My2dPoint P = new My2dPoint(2,4);
Console.WriteLine(P);
```

My2dPoint x:2 y:4

Write/WriteLine work *out-of-the-box* because structures (value types) inherit from *System.Object* object

21

Structures: operators ==, !=, *Equals()* method … (#14/23)

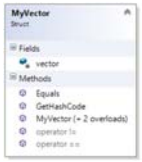
operator == overloading allows support for reference fields

```
struct MyVector
{
  private double[] vector;

  public MyVector(int newSize, double defaultValue = 0.0)
  { /* ... */ }
  public MyVector(params double[] newValues)
  { /* ... */ }
  public MyVector(MyVector src)
  { /* ... */ }

  public static bool operator ==(MyVector mv1, MyVector mv2)
  { /* ... */ }
  public static bool operator !=(MyVector mv1, MyVector mv2)
  { /* ... */ }
  public override bool Equals(object obj)
  { /* ... */ }
  public override int GetHashCode()
  { /* ... */ }
}
```

data: reference type

constructors

operators & methods set to override "==" operator



22

Structures: operators ==, !=, *Equals()* method … (#15/23)

overloading == operator allows considering reference fields (together with == and != operators *Equals()* and possibly *GetHashCode*() should be overridden)

```
public MyVector(int newSize, double defaultValue = 0.0)
{
  vector = new double[newSize];
  for (int i = 0; i < this.vector.Length; ++i) this.vector[i] = defaultValue;
}
```

- creating an object containing a vector of given size (*newSize*)
- by default vector is initialised with zeroes (*0.0*) or with *defaultValue* argument, if provided

```
public MyVector(params double[] newValues)
{
  vector = new double[newValues.Length];
  for (int i = 0; i < this.vector.Length; ++i) this.vector[i] = newValues[i];
}
```

creating a vector of given, in the call argument, number of elements

```
public MyVector(MyVector src)
{
  this.vector = (double[])src.vector.Clone();
}
```

creating a deep copy of the source vector

23

Structures: operators ==, !=, *Equals()* method … (#16/23)

```
public static bool operator ==(MyVector mv1, MyVector mv2)
{
  if (ReferenceEquals(mv1.vector, mv2.vector)) return true;
  if (mv1.vector.Length != mv2.vector.Length) return false;
  for (int i = 0; i < mv1.vector.Length; ++i)
    if (mv1.vector[i] != mv2.vector[i]) return false;
  return true;
}
```

this implementation is slightly different than the class version:
- it is pointless to call *ReferenceEquals(mv1, mv2)* – structures are boxed (in different boxes) and therefore never equal, the only thing to check is whether the structures do not contain shallow copy (shared array) of reference variables,
- if both vectors are equal in length then we have to compare element by element, if any pair differs then the vectors are different

```
public static bool operator !=(MyVector mv1, MyVector mv2)
{
  return !(mv1 == mv2);
}
```


24

Object-oriented programming

Structures: operators ==, !=, *Equals()* method … (#17/23)

```
public override bool Equals(object obj)
{
  if(obj != null)
    return (this == (MyVector)obj);
  else return false;
}
public override int GetHashCode()
{
  long h = 1;
  for (int i = 0; i < vector.Length; i++)
  {
    long DoubleAsInt = BitConverter.DoubleToInt64Bits(vector[i]);
    h = 31 * h + DoubleAsInt;
  }
  h ^= (h >> 20) ^ (h >> 12);
  return (int)(h ^ (h >> 7) ^ (h >> 4));
}
```

- is the object properly boxed?
- let's use previously overloaded "==" operator

hash function implementation inspired by Java

Object-oriented programming (25/32)

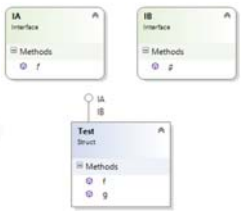
25

Object-oriented programming

Structures: interface implementation (#18/23)

structures cannot inherit from structures/classes but they can inherit from interfaces (implement interfaces)

```
interface IA
{
    int f(int a);
}
interface IB
{
    int g(float a);
}
struct Test : IA, IB
{
    public int f(int a) { return 0; }
    public int g(float a) { return 0; }
}
```



Object-oriented programming (26/32)

26

Object-oriented programming

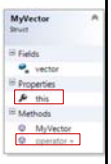
Structures: operator overloading, indexer (#19/23)

```
struct MyVector
{
  private double[] vector;
  public MyVector(int newSize, double defaultValue = 0.0)
  {
    vector = new double[newSize];
    for (int i = 0; i < this.vector.Length; ++i) this.vector[i] = defaultValue;
  }
  public static MyVector operator +(MyVector mv1, MyVector mv2)
  {
    // no null references tests – useless for structures
    if (mv1.vector.Length != mv2.vector.Length)
      throw new ArgumentException("Incompatible vector dimensions");
    MyVector tmv = new MyVector(mv1.vector.Length);
    for (int i = 0; i < tmv.vector.Length; ++i)
        tmv.vector[i] = mv1.vector[i] + mv2.vector[i];
    return tmv;
  }
  public double this[int index]
  {
    get {  return this.vector[index]; }
    set {  this.vector[index] = value; }
  }
}
```

constructor

overloaded '+' operator

"[]" operator/property (indexer)



Object-oriented programming (27/32)

27

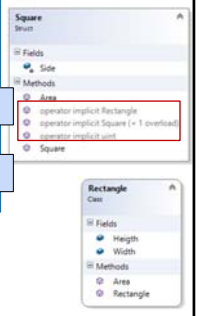
Object-oriented programming

Structures: converters (#20/23)

```
struct Square
{
  uint Side;
  public Square(uint newSide) { this.Side = newSide; }
  public double Area() { return this.Side * this.Side; }

  public static implicit operator Square(Rectangle s)
    { return new Square((s.Width + s.Heigth) / 2); }
  public static implicit operator Rectangle(Square s)
    { return new Rectangle(s.Side, s.Side); }

  public static implicit operator uint (Square s)
    { return s.Side;}
  public static implicit operator Square(uint s)
    { return new Square(s); }
}

class Rectangle
{
  public uint Width, Heigth;
  public Rectangle(uint newHeight, uint newWidth)
  { /* ... */ }
  /* ... */
}
```

conversions *struct* ⇔ *class*

conversions *struct* ⇔ *uint*



Object-oriented programming (28/32)

28

Object-oriented programming

Structures: passing by value (#21/23)

passing structures by value (copy):

```
struct S1
{
  public int x, y;
}
...
void P2(S1 a)
{
  a.x = 5;
}
...
{
  S1 b;
  b.x = b.y = 3; // all fields must be initialized
  Console.WriteLine("b.x: " + b.x);
  P2(b);
  Console.WriteLine("b.x: " + b.x);
}
```

type definition

a: the formal parameter (local copy of value/variable)

modification of the local copy

caution required if fields are reference type

`b.x: 3`

`b.x: 3`

invocation of *P2*() method, passing a copy of variable *b*

Object-oriented programming (29/32)

29

Object-oriented programming

Structures: passing by reference (#22/23)

passing structures by reference (*ref* mode):

```
struct S1
{
  public int x, y;
}
...
void P2(ref S1 a)
{
  a.x = 5;
}
...
{
  S1 b;
  b.x = b.y = 3; // all fields must be initialized
  Console.WriteLine("b.x: " + b.x);
  P2(ref b);
  Console.WriteLine("b.x: " + b.x);
}
```

type definition

a: the formal parameter (the reference to the variable)

modification of the original variable using the reference

`b.x: 3`

`b.x: 5`

invocation of *P2*() method, passing the reference to variable *b*

Object-oriented programming (30/32)

30

Structures: passing by output (#23/23)

passing structures by output (*out* mode):

```
struct s1
{
  public int x, y;
}
...
void P2(out s1 a)
{
  //  Console.WriteLine(a.x); <- use of unassigned out parameter
  a.x = a.y = 5; // required assignment to all components
}
...
{
  s1 b;
  b.x = 3;  // b.x may be used, b.y – may not
  Console.WriteLine("b.x: " + b.x);  b.x: 3
  P2(out b);
  Console.WriteLine("b.x: " + b.x);  b.x: 5
}
```

type definition

a: the formal parameter (the reference to the variable)

invocation of *P2*() method, passing the reference to the variable *b*

31

Structures: summary

- structures are value types (stack), classes - reference types (heap), assignment operator (=) in the case of structures copies all fields, in the case of classes - references to objects,
- one cannot define custom default (instance) constructor in structures – regardless of the defined constructors, an auto-implemented default constructor is always available,
- the structure constructor is used to initialise members (does not create a new structure, except for temporary data, e.g. returned from the method: *return new StructType ()*)
- structures are always descendants of *System.ValueType*, which inherits from *System.Object* but cannot have child classes (like the *sealed* final classes),
- structures cannot inherit from other structures/classes (excluding *System.ValueType* and *System.Object*) but can implement interfaces,
- structure fields are not automatically initialised,
- structures cannot create recursive data structures (boxing is required),
- C# value types are implemented as structures, e.g. *uint* is:

```
public struct UInt32 : IComparable, IFormattable, IConvertible,
                       IComparable<UInt32>, IEquatable<UInt32> { /* ... */ }
```


32