Object-oriented programming #6
Operator overloading, indexer, boxing/unboxing

Wojciech Complak

Institute of Computing Science
Faculty of Computing
Poznan University of Technology

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

1

Lecture content

- **operator overloading**
- **indexer (operator [])**
- **boxing/unboxing**

2

Operator overloading: case study (#1/31)

Case:
design a class performing calculations on *double* type vectors,
support standard C# arithmetic operators
e.g. instead of a call:

```
Vector3 = AddVector(Vector1, Vector1)
```

support more natural (and convenient) form:

```
Vector3 = Vector1 + Vector1
```

Requirements:
- **the ability to create vectors,**
- **output to Console using *Write()/WriteLine()* methods**
- **availability of arithmetic operations of addition (+), compound assignment (+=), and more ...**

3

Operator overloading: class design (#2/31)

data:

```
class MyVector
{
  private double[] vector;
}
```

private field to store vector elements (vector = single-dimensional array)

constructor #1:

```
private MyVector() { }
```

„empty" vector is not viable:
~~MyVector mv = new MyVector();~~

4

Operator overloading: class design (#3/31)

constructor #2:

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

constructor supports:
- creating an object containing a vector of given size *(newSize)*
- initialising the vector with *0.0* (by default) or with argument *defaultValue*

```
MyVector mv = new MyVector(5,3.0); // [ 3, 3, 3, 3, 3]
```

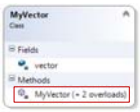
(basically, we can now opt out of the private default constructor (#1))

5

Operator overloading: class design (#4/31)

constructor #3:

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



constructor supports creating a vector based on given arguments:

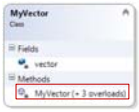
```
MyVector mv = new MyVector(1.0, 2.0, 3.0, 4.0, 5.0);
                          // [ 1, 2, 3, 4, 5]
```


6

Operator overloading: class design (#5/31)

constructor #4:

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



copy constructor creates deep source object copy:

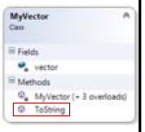
```
MyVector mv = new MyVector(1.0, 2.0, 3.0, 4.0, 5.0);
                          // [ 1, 2, 3, 4, 5]
MyVector mv1 = new MyVector(mv);
                          // [ 1, 2, 3, 4, 5]
```


7

Operator overloading: *ToString()* method (#6/31)

ToString() method:

```
public override string ToString()
{
 string ReturnString = "[";
 for (int i = 0; i < this.vector.Length; ++i)
 {
   ReturnString += " " + this.vector[i];
   if (i == this.vector.Length - 1) ReturnString += "]";
   else ReturnString += ",";
 }
 return ReturnString;
}
```



useful to output *MyVector* class objects with *Write*/*WriteLine* methods of the *Console* object, example:

```
MyVector mv = new MyVector(1.0, 2.0, 3.0, 4.0, 5.0);
Console.WriteLine(mv);          // [ 1, 2, 3, 4, 5]
```


8

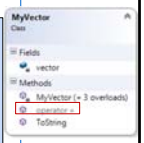
Operator overloading: + operator (#7/31)

overloading '+' operator

mathematical foundations: $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}$

```
public static MyVector operator +(MyVector left, MyVector right)
```

- operator overloading is implemented by a method that must be public and static, it's not C++: C# does not support virtual operators, operators must be defined within the class
- after the *operator* keyword, we specify which C# operator we want to overload (you cannot add new operators, you cannot change bindings and priorities),
- in round brackets we specify the arguments on which the operator is to operate (in the same number as the original operator we want to overload)
- in the example: overloaded binary addition operator '+' applied to two *MyVector* objects, the result is also a *MyVector* object



9

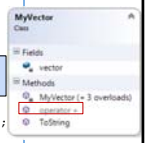
Operator overloading: + operator (#8/31)

overloading '+' operator

mathematical foundations: $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}$

```
public static MyVector operator +(MyVector left, MyVector right)
{
  if (((object)left == null) || ((object)right == null))
    throw new ArgumentException("null vector operation");
  if (left.vector.Length != right.vector.Length)
    throw new ArgumentException("Incompatible vector dimensions");
  MyVector tmv = new MyVector(left.vector.Length);
  for (int i = 0; i < tmv.vector.Length; ++i)
     tmv.vector[i] = left.vector[i] + right.vector[i];
  return tmv;
}
```

- what to do if one or both arguments are *null* references? no answer makes sense for vectors, let's throw an exception



10

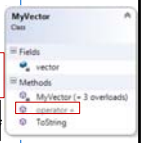
Operator overloading: + operator (#9/31)

overloading '+' operator

mathematical foundations: $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}$

```
public static MyVector operator +(MyVector left, MyVector right)
{
  if (((object)left == null) || ((object)right == null))
    throw new ArgumentException("null vector operation");
  if (!left.GetType().Equals(right.GetType()) ||
      !left.GetType().Equals(typeof(MyVector)))
    throw new ArgumentException("Incompatible types");
  for (int i = 0; i < tmv.vector.Length; ++i)
     tmv.vector[i] = left.vector[i] + right.vector[i];
  return tmv;
}
```

- the class is not final (*sealed*) so let's verify if both (left and right) types are the same and equal to *MyVector*



11

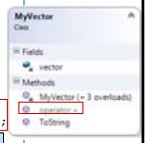
Operator overloading: + operator (#10/31)

overloading '+' operator

mathematical foundations: $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}$

```
public static MyVector operator +(MyVector left, MyVector right)
{
  if (((object)left == null) || ((object)right == null))
    throw new ArgumentException("null vector operation");
  if (!left.GetType().Equals(right.GetType()) ||
      !left.GetType().Equals(typeof(MyVector)))
    throw new ArgumentException("Incompatible types");
  if (left.vector.Length != right.vector.Length)
    throw new ArgumentException("Incompatible vector dimensions");
  return tmv;
}
```

- mathematical definition shows that only vectors of the same size can be added to each other, if they have different lengths let's throw an exception



12

Operator overloading: + operator (#11/31)

overloading '+' operator

mathematical foundations: $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}$

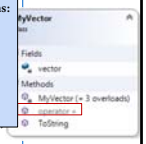
- result: vector of the same length, containing sums of elements in the appropriate positions:

```
MyVector mv1 = new MyVector(1.5, 2.5);
Console.WriteLine(mv1);          // [ 1,5, 2,5]

MyVector mv2 = new MyVector(1.0, 3.0);
Console.WriteLine(mv2);          // [ 1, 3]

MyVector mv3 = mv1 + mv2;
Console.WriteLine(mv3);          // [ 2,5, 5,5]
```

```
  MyVector tmv = new MyVector(left.vector.Length);
  for (int i = 0; i < tmv.vector.Length; ++i)
     tmv.vector[i] = left.vector[i] + right.vector[i];
  return tmv;
}
```



13

Operator overloading: += operator (#12/31)

overloading "+=" operator

```
public static MyVector operator += (MyVector mv)
{

}
```

- compound assignment operators (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=) cannot be directly overloaded (contrary to C++), but it is enough to overload appropriate binary operator:
  the compiler will automatically convert operation like:
  `a += b`
  to:
  `a = a + b`
- the assignment operator (=) cannot be overloaded in C# (again, contrary to C++) - this would be incompatible with the semantics of operating on references, instead of on class objects (reference types would start behaving as value types)

14

Operator overloading: ++ operator (#13/31)

overloading "++" operator

never overload operators by giving them a non-standard (not obvious) interpretation (after all, we are adding operators to simplify the use of the class) - this solution brings more harm than good (e.g. adding !? prints to the screen ... because we add text, and what subtraction is to do ...)

DO NOT overload operators if it makes it difficult to understand the functionality of the type

mathematical interpretation: $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + + \Leftrightarrow \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

☞ C# does not support separate pre/post in/decrementation (contrary to C++), value returned by the operator is handled automatically

```
public static MyVector operator ++(MyVector mv)
```

- method implementing operator is *public* and *static*,
- receives single argument– *MyVector* class object,
- *MyVector* class object is the result,
- in the example: operator ++ adds unit vector

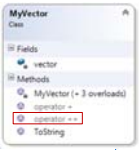
```
  return mv + tmv;
}
```


15

Operator overloading: ++ operator (#14/31)

overloading "++" operator

mathematical interpretation: $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + + \Leftrightarrow \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$




```
public static MyVector operator ++(MyVector mv)
{
  if ((object)mv == null) throw new ArgumentException("null vector operation");
  if( !mv.GetType().Equals(typeof(MyVector)))
    throw new ArgumentException("Incompatible type");
  MyVector tmv = new MyVector(mv.vector.Length, 1.0);
  return mv + tmv;
}
```

- if argument is *null* or of incorrect type let's throw an exception,
- else:
  create unit vector of required size,
  use previously overloaded '+' operator to compute result

16

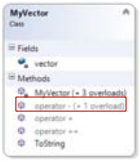
Operator overloading: - operator (#15/31)

unary and binary versions (if present) of an operator are to be overloaded separately

mathematical interpretation:

binary minus: $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 - b_1 \\ a_2 - b_2 \\ a_3 - b_3 \end{bmatrix}$

unary minus: $-\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} -a_1 \\ -a_2 \\ -a_3 \end{bmatrix}$



☞ two separate methods:

- for binary minus:
  public static MyVector operator -(MyVector mv1, MyVector mv2) { /* ... */ }
- for unary minus:
  public static MyVector operator -(MyVector mv) { /* ... */ }

17

Operator overloading: - operator(s) (#16/31)

overloading binary '-' operator

```
public static MyVector operator -(MyVector left, MyVector right)
{
  if (((object)left == null) || ((object)right == null))
    throw new ArgumentException("null vector operation");
  if (!left.GetType().Equals(right.GetType()) ||
      !left.GetType().Equals(typeof(MyVector)))
    throw new ArgumentException("Incompatible types");
  if (left.vector.Length != right.vector.Length)
    throw new ArgumentException("Incompatible vector dimensions");
  MyVector tmv = new MyVector(left.vector.Length);
  for (int i = 0; i < tmv.vector.Length; ++i)
     tmv.vector[i] = left.vector[i] - right.vector[i];
  return tmv;
}
```

implementation similar to adding vectors

18

Operator overloading: - operator(s) (#17/31)

overloading unary '-' operator

```
public static MyVector operator -(MyVector mv)
{
  if ((object)mv == null) throw new ArgumentException("null vector operation");
  if (!mv.GetType().Equals(typeof(MyVector)))
    throw new ArgumentException("Incompatible type");
  MyVector tmv = new MyVector(mv.vector.Length);
  for (int i = 0; i < tmv.vector.Length; ++i) tmv.vector[i] = -mv.vector[i];
  return tmv;
}
```

return new vector of the same size with negated respective values

19

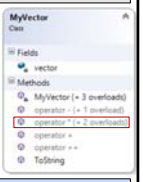
Operator overloading: * operator(s) (#18/31)

overloading an operator for different argument types

mathematical interpretation:

scalar product: $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \sum_{i=1}^{n} a_i * b_i$

vector scaling: $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} * c = \begin{bmatrix} c * a_1 \\ c * a_2 \\ c * a_3 \end{bmatrix}$



☞ two methods:

- for scalar product:
  `public static double operator *(MyVector mv1, MyVector mv2) { /* … */ }`
- for vector scaling:
  `public static MyVector operator *(MyVector mv1, double c) { /* … */ }`
  note: if commutative operator is required add second version:
  `public static MyVector operator *(double c, MyVector mv1) { /* … */ }`

np.:

```
MyVector mv1 = new MyVector(1.0, 2.0, 3.0, 4.0, 5.0);
MyVector mv2 = mv1 * 1;
MyVector mv3 = 2 * mv1;
```


20

Operator overloading: * operator(s) (#19/31)

overloading '*' operator

```
public static MyVector operator *(MyVector mv, double c)
{
  if ((object)mv == null) throw new ArgumentException("null vector operation");
  if (!mv.GetType().Equals(typeof(MyVector)))
    throw new ArgumentException("Incompatible type");
  MyVector tmv = new MyVector(mv.vector.Length);
  for (int i = 0; i < tmv.vector.Length; ++i) tmv.vector[i] = mv.vector[i] * c;
  return tmv;
}
```

vector scaling(*MyVector*, *double*)

```
public static MyVector operator *(double c, MyVector mv1)
{
  return mv1 * c;
}
```

vector scaling(*double*, *MyVector*) – let's use existing implementation of '*' operator

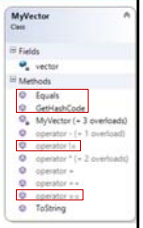
commutativeness

```
public static double operator *(MyVector left, MyVector right)
{
  if (((object)left == null) || ((object)right == null))
    throw new ArgumentException("null vector operation");
  if (!left.GetType().Equals(right.GetType()) || !left.GetType().Equals(typeof(MyVector)))
    throw new ArgumentException("Incompatible types");
  if (left.vector.Length != right.vector.Length)
        throw new ArgumentException("Incompatible vector dimensions");
  double ReturnValue = 0;
  for (int i = 0; i < left.vector.Length; ++i) ReturnValue += left.vector[i] * right.vector[i];
  return ReturnValue;
}
```

vectors scalar product

21

Operator overloading: == operator (#20/31)



note:

- C# requires overloading of logical operators in pairs:
  "==" and "!="
  "<" and ">"
  "<=" and ">="
- if == operator is overloaded (and != too, of course) *Equals()* and *GetHashCode()* methods should be overridden

☞ four methods:

- for "==" operator:
  `public static bool operator ==(MyVector mv1, MyVector mv2) {/*…*/}`
- for "!=" operator:
  `public static bool operator !=(MyVector mv1, MyVector mv2) {/*…*/}`
- *Equals()*:
  `public override bool Equals(object obj) {/*…*/}`
- *GetHashCode()*:
  `public override int GetHashCode() {/*…*/}`

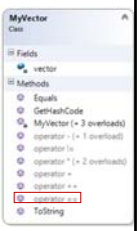
22

Operator overloading: == operator (#21/31)

overloading "==" operator

```
public static bool operator ==(MyVector left, MyVector right)
{
  if (ReferenceEquals(left, right)) return true;
  if (((object)left == null) || ((object)right == null))
    return false;
  if (left.GetType().Equals(right.GetType()) &&
      left.GetType().Equals(typeof(MyVector)))
    return VectorsEqual(left.vector, right.vector);
  else return false;
}
```

- if both references are equal then *true*
- *null* = *null* (C# convention)



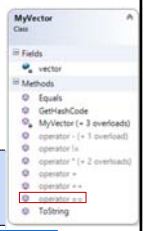
23

Operator overloading: == operator (#22/31)

overloading "==" operator

```
public static bool operator ==(MyVector left, MyVector right)
{
  if (ReferenceEquals(left, right)) return true;
  if (((object)left == null) || ((object)right == null))
    return false;
  if (left.GetType().Equals(right.GetType()) &&
      left.GetType().Equals(typeof(MyVector)))
    return VectorsEqual(left.vector, right.vector);
  else return false;
```

- if one is *null*, and the other is not, then – not equal,
- casting to (*object*) avoids infinite recursion (or use *ReferenceEquals()* directly)
- check types and use auxiliary *VectorsEqual()* method




```
private static bool VectorsEqual(double[] v1,double[] v2)
{
  if (v1.Length != v2.Length) return false;
  for (int i = 0; i < v1.Length; ++i)if (v1[i] != v2[i]) return false;
  return true;
}
```

- if lenghts of vector are not equal then false,
- if lengths are equal then compare contents

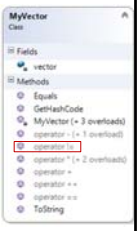
24

Operator overloading: != operator (#23/31)

overloading "!=" operator

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

let's use previously defined == operator



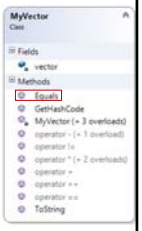
25

Operator overloading: *Equals()* method (#24/31)

overriding *Equals()* method

```
public override bool Equals(object other)
{
  // if(this == null) ...
  ...er).vector);
  else return false;
}
```

this test is usually not necessary – C# will protect programmer against such situation, problem (this == *null*) may be caused by reflection or invocation from other language



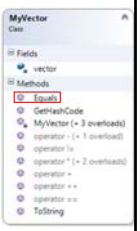
26

Operator overloading: *Equals()* method (#25/31)

overriding *Equals()* method

```
public override bool Equals(object other)
{
  // if(this == null) ...
  if ((object)other == null) return false;
  ...
    return VectorsEqual(this.vector, ((MyVector)other).vector);
  else return false;
}
```

if other object is *null* then return not equal



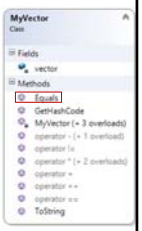
27

Operator overloading: *Equals()* method (#26/31)

overriding *Equals()* method

```
public override bool Equals(object other)
{
  // if(this == null) ...
  if ((object)other == null) return false;
  if (ReferenceEquals(this, other)) return true;
  ...other).vector);
  else return false;
}
```

if both references are equal (same object) then return equal



28

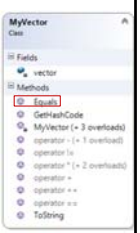
Operator overloading: *Equals()* method (#27/31)

overriding *Equals()* method

```
public override bool Equals(object other)
{
  // if(this == null) ...
  if ((object)other == null) return false;
  if (ReferenceEquals(this, other)) return true;
  if (this.GetType().Equals(other.GetType()))
```

things to consider when creating a hierarchy:

- is the type of argument correct – if it is a child object can we compare two objects on different levels of the same hierarchy?
- how (if) to compare inherited parts (*base.Equals*()) ?



29

Operator overloading: *Equals()* method (#28/31)

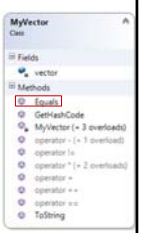
overriding and overloading *Equals()* method

```
public override bool Equals(object other)
{
  // if(this == null) ...
  if ((object)other == null) return false;
  if (ReferenceEquals(this, other)) return true;
  if (this.GetType().Equals(other.GetType()))
    return VectorsEqual(this.vector, ((MyVector)other).vector);
  else return false;
}
```

let's use already implemented *VectorsEqual()* method

☞ for efficiency reasons it is useful to implement custom non-virtual overloaded *Equals*() method:

```
public bool Equals(MyVector mv)
{
  if ((object)mv == null) return false;
  if (ReferenceEquals(this, mv)) return true;
  if (this.GetType().Equals(mv.GetType()))
    return VectorsEqual(this.vector, mv.vector);
  else return false;
}
```

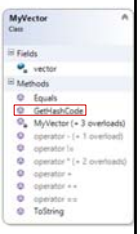


30

Operator overloading: *GetHashCode()* method (#29/31)

overriding *GetHashCode()* method

```
public override int GetHashCode()
{
 // implementation inspired by Java solutions
 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));
}
```



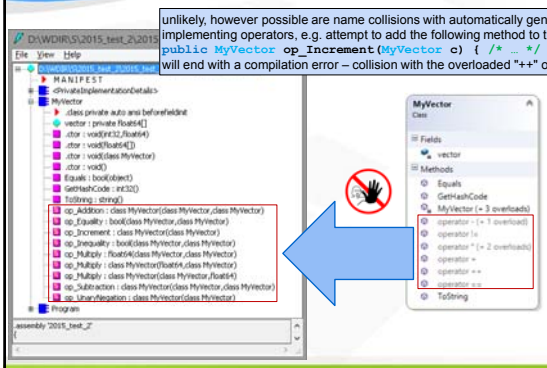
the hash function should :
- conform to simple uniform hashing assumption: a randomly selected key is mapped with equal probability to each of the *m* values, regardless of the mapping of other keys to minimize collisions
- be easy to calculate,

the implementation shown is:
- complex modular hashing (division remainder):
  H(k) = k mod m

31

Operator overloading: implementation (#30/31)

unlikely, however possible are name collisions with automatically generated methods implementing operators, e.g. attempt to add the following method to the class:
`public MyVector op_Increment(MyVector c) { /* … */ }`
will end with a compilation error – collision with the overloaded "++" operator



32

Operators summary (#31/31)

Operators	Overloadability
`+, -, !, ~, ++, --, true, false`	These unary operators can be overloaded.
`+, -, *, /, %, &, |, ^, <<, >>`	These binary operators can be overloaded.
`==, !=, <, >, <=, >=`	Binary comparison operators can be overloaded but they must be overloaded in pairs (== and !=, < and >, <= and >=)
`&&, ||`	Conditional logical operators cannot be overloaded. However, if a type with the overloaded *true* and *false* operators also overloads the & or | operator, the && or || operator, respectively, can be evaluated for the operands of that type.
`[]`	Element access is not considered an overloadable operator, but you can define an indexer.
`()`	The cast operator cannot be overloaded, but you can define new conversion operators.
`+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`	Compound assignment operators cannot be explicitly overloaded. However, when you overload a binary operator, the corresponding compound assignment operator, if any, is also implicitly overloaded.
`=, ., ?:, ->, new, is, sizeof, typeof`	These operators cannot be overloaded.

33

Inheritance and operator overloading (#1/3)

```
class Parent // : Object
{
  private int a;
  public Parent(int newA) { this.a = newA; }
  public static bool operator ==(Parent left, Parent right)
  {   if (ReferenceEquals(left, right)) return true; // null == null
      if (((object)left == null) || ((object)right == null)) return false;
      return (left.Equals(right));
  }
  public static bool operator !=(Parent left, Parent right)
  {   return !(left == right);
  }
  public override bool Equals(object other)
  {   if (((object)other == null) ||
          (this.GetType().Equals(other.GetType()) == false)) return false;
      return this.a == ((Parent)other).a;
  }
  public override int GetHashCode() {  return this.a; }
}
```

if this object is to have descendants proper (non-static) method must invoked -> virtual *Equals()* method

34

Inheritance and operator overloading (#2/3)

```
class Child : Parent
{ private int b;
  public Child(int newA, int newB) : base(newA)  {  this.b = newB; }
  public static bool operator ==(Child left, Child right)
  {   if (ReferenceEquals(left, right)) return true;
      if (((object)left == null) || ((object)right == null)) return false;
      return (left.Equals(right));
  }
  public static bool operator !=(Child left, Child right)
  {  return !(left == right);
  }
  public override bool Equals(object other)
  {   if (((object)other == null) ||
          (this.GetType().Equals(other.GetType()) == false)) return false;
      return base.Equals((Parent)other) && (this.b == ((Child)other).b);
  }
  public override int GetHashCode()
  {   // funkcja pary Cantora
      int k1 = base.GetHashCode(), k2 = this.b;
      return (((k1 + k2) * (k1 + k2 + 1)) / 2) + k2;
  }
}
```

in child class: call to base class *Equals()* and comparison of added members

35

Inheritance and operator overloading (#3/3)

```
Parent p1 = new Parent(1), p2 = new Parent(1), p3 = new Parent(2);
Console.WriteLine(p1 == p2); // Parent == Parent (true)
Console.WriteLine(p2 == p3); // Parent == Parent (false)

Child c1 = new Child(1, 1), c2 = new Child(1, 1), c3 = new Child(1, 2);
Console.WriteLine(c1 == c2);  // Child == Child (true)
Console.WriteLine(c2 == c3);  // Child == Child (false)
Console.WriteLine(p2 == c3);  // Parent == Child (false)
Console.WriteLine(c2 == p2);  // Child == Parent (false)

p1 = c1;
p2 = c2;
p3 = c3;
Console.WriteLine(p1 == p2);  // Parent(Child) == Parent(Child), (true) O.K.
Console.WriteLine(p2 == p3);  // Parent(Child) == Parent(Child), (false) O.K.
Console.WriteLine(p2 == c3);  // Parent(Child) == Child,         (false), O.K.
Console.WriteLine(c2 == p2);  // Child == Parent(Child),         (true), O.K.
Console.WriteLine(c2 == c2);  // Child == Parent(Child),         (true), O.K.
```

incompatible types

thanks to virtual *Equals()* method

36

Indexer (operator [] overloading) (#1/10)

element access operator is a comfortable way of iterating through indexable data structures

C# does not support overloading of operator [] (by a static method) but similar effect is achieved using an indexer

Case:
design a class supporting dynamic array of non-negative integers (*uint*),
implement support for setting and getting vector elements similar to a regular C# array

data:

```
class UintDynamicArray
{
  uint[] array;
  uint defaultIncrement,
       defaultValue;
}
```

- vector of element values
- default increment of vector growth
- default initial value for elements in the array

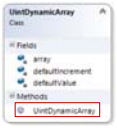
37

Indexer (operator [] overloading) (#2/10)

constructor:

```
public UintDynamicArray(int initialSize = 10,
                        uint defaultIncrement = 10,
                        uint defaultValue = 0)
{
  array = new uint[initialSize];
  this.defaultIncrement = defaultIncrement;
  this.defaultValue = defaultValue;
  for (int i = 0; i < array.Length; ++i)
    array[i] = this.defaultValue;
}
```

constructor arguments are:
- initial size of the array (default 10)
- array size growth step (default 10)
- initial values of vector elements (default 0)



38

Indexer (operator [] overloading) (#3/10)

constructor:

```
public UintDynamicArray(int initialSize = 10,
                        uint defaultIncrement = 10,
                        uint defaultValue = 0)
{
  array = new uint[initialSize];
  this.defaultIncrement = defaultIncrement;
  this.defaultValue = defaultValue;
  for (int i = 0; i < array.Length; ++i)
    array[i] = this.defaultValue;
}
```

constructor:
- creates a new array
- stores array growth increment
- stores array elements initial value
- fills new array with default values

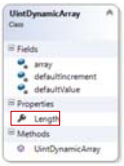
39

Indexer (operator [] overloading) (#4/10)

Length property

```
public int Length
{
  get { return array.Length; }
}
```

Length property (if it is to be available) must be implemented



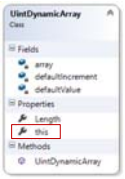
40

Indexer (operator [] overloading) (#5/10)

indexer

```
public uint this[int index]
{
 get
 {
  i                                  " +
                                     gth -
  return this.array[index];
 }
 set
 {
  if (index < 0)
   throw new IndexOutOfRangeException("Index < 0");
  if (index >= array.Length)growArray(index + 1);
  this.array[index] = value;
 }
}
```

- indexer implementation is similar to a property except keyword *this* and square brackets [],
- indexer, like a property, can be read-only (*get* only) and write-only (*set* only),
- get/set may have different access modifiers (*public*, *protected*, *private*)



41

Indexer (operator [] overloading) (#6/10)

indexer

```
public uint this[int index]
{
 get
 {
  if ((index < 0) || (index >= array.Length))
   throw new IndexOutOfRangeException("Attempt to read index " +
            index + " outside correct range <0," + (array.Length - 1) + ">");
  return this.array[index];
 }
 se
 {
  if (index < 0)
   throw new IndexOutOfRangeException("Index < 0");
  if (index >= array.Length)growArray(index + 1);
  this.array[index] = value;
 }
}
```

while reading an element verify if index is in designated range and if it is not throw an exception (array is not grown automatically)

42

Indexer (operator [] overloading) (#7/10)

indexer

```
public uint this[int index]
{
 get
 {
  if ((index < 0) || (index >= array.Length))
   throw new IndexOutOfRangeException("Attempt to read index " +
            index + " outside correct range <0," + (array.Length - 1) + ">");
  return this.array[index];
 }
 set
 {
  if (index < 0)
   throw new IndexOutOfRangeException("Index < 0");
  if (index >= array.Length)growArray(index + 1);
  this.array[index] = value;
 }
}
```

while writing check if index is not negative (if it is then throw an exception),
next, if index is outside current array length then request array growth and write the element

43

Indexer (operator [] overloading) (#8/10)

indexer

```
public uint this[int index]
{
 get
 {
  if ((index < 0) || (index >= array.Length))
   throw new IndexOutOfRangeException("Próba odczytu indeksu " +
            index + " spoza prawidłowego zakresu <0," + (array.Length - 1) + ">");
  return this.array[index];
 }
 set
 {
  if (index < 0)
   throw new IndexOutOfRangeException("Index < 0");
  if (index >= array.Length)growArray(index + 1);
  this.array[index] = value;
 }
}
```

```
UintDynamicArray MyArray = new UintDynamicArray(5, 1000, 5);
MyArray[i] = 3;
Console.WriteLine(MyArray[3]);
```

create array,
read and write using the indexer

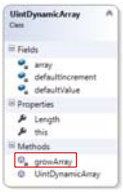
44

Indexer (operator [] overloading) (#9/10)

growArray() method

```
private void growArray(int newSize)
{
  int MinIncrement = this.array.Length + (int)this.defaultIncrement;
  if (newSize < MinIncrement) newSize = MinIncrement;
  uint[] tarray = new uint[newSize];
  Array.Copy(array, tarray, array.Length);
  for (int i = array.Length; i < newSize; ++i) tarray[i] = this.defaultValue;
  array = tarray;
}
```

to enlarge array:
- new size must be at least *defaultIncrement* bigger,
- create new array,
- copy existing elements,
- initialise the new part of the array with default value



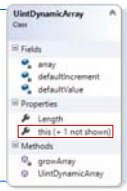
45

Indexer (operator [] overloading) (#10/10)

indexer

```
public uint this[string index]
{
  get
  {
    return this.array[int.Parse(index)];
  }
  set
  {
    this.array[int.Parse(index)] = value;
  }
}
```

indexer can be overloaded




```
public uint this[int a, int b, int c]
{
  get { /* ... */ }
  set { /* ... */ }
}
```

indexer can be multidimensional

46

Boxing/unboxing

boxing is the process of creating a reference (*System.Object* class object) pointing to new object created on the heap, during boxing source value variable is copied to the new object, this operation is necessary to treat value types as objects (for example to store in a collection),
the reverse process is performed while unboxing

```
int MyInt = 1;
object BoxedValue = MyInt;        // boxing
WhatsInTheBox(BoxedValue);        // what's in the box ("it's an int")
int MyNewInt = (int)BoxedValue;   // unboxing requires explicit casting

uint MyUint = 2;
BoxedValue = MyUint;              // boxing into the same box
WhatsInTheBox(BoxedValue);        // what's in the box ("System.UInt32")
MyNewInt = (int)BoxedValue;       // unboxing error: InvalidCastException
                                  // incorrect type
```

```
void WhatsInTheBox(object obj)
{
  if (obj is int) Console.WriteLine("it's an int");
  else Console.WriteLine(obj.GetType());
}
```

information about value type is stored in the box and can be checked dynamically

47