Object-oriented programming #5
Serialization, generics

Wojciech Complak

Institute of Computing Science
Faculty of Computing
Poznan University of Technology

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

1

Lecture content

- **serialization**
- **generics**

2

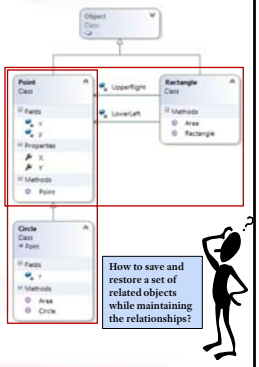
Serialization

```
class Point
{
 private double x, y;
 public Point(double newX, double newY){ x = newX; y = newY; }
 public double X { get { return x; } }
 public double Y { get { return y; } }
}
class Circle : Point
{
 private double r;
 public Circle(double newX, double newY, double newR)
     : base(newX, newY) { r = newR; }
 public double Area() { return Math.PI * r * r; }
}
class Rectangle
{
 private Point LowerLeft, UpperRight;
 public Rectangle(double x1,double y1,double x2,double y2)
 {
   LowerLeft = new Point(x1, y1);
   UpperRight = new Point(x2, y2);
 }
  public double Area()
  { return Math.Abs((LowerLeft.X-UpperRight.X)*(LowerLeft.Y-UpperRight.Y)); }
}
```

base class

"is-a" inheritance

"has-a" relationship



How to save and restore a set of related objects while maintaining the relationships?

3

Serialization

```
[Serializable]
class Point
{
 private double x, y;
 /* ... */
}
[Serializable]
class Circle : Point
{
 private double r;
 /* ... */
}
[Serializable]
class Rectangle
{
 private Point LowerLeft, UpperRight;
 /* ... */
}
```

- flag each (flag is not inherited) class that has to be serializable with the *[Serializable]* attribute, an attempt to serialize an unflagged object will result in a *SerializationException* exception,
- if a field is not to be serialized (e.g. temporary buffer), to save time and data size mark the field as *[NonSerialized]*,
- choose the serializer, three serializers are available as standard:
  - *BinaryFormatter* - stores data in a binary stream in a compact form, can save/restore private fields and properties,
  - *SoapFormatter* - wraps serialized data in XML format in a SOAP (Simple Object Access Protocol) envelope in a standard form for communication with web services, can serialize private fields and properties,
  - *XmlSerializer* - saves objects in pure XML, only public fields and properties are supported

4

Serialization: *BinaryFormatter*

```
Circle c = new Circle(2, 2, 4);
Rectangle r = new Rectangle(1, 1, 5, 3);

BinaryFormatter binFormattter = new BinaryFormatter();

using (Stream fStream =
        new FileStream("mydata.dat", FileMode.Create, FileAccess.Write))
  {
    binFormattter.Serialize(fStream, c);
    binFormattter.Serialize(fStream, r);
  }
```

create new binary formatter

- open/create a stream to write
- serialize (store) objects to the stream using *Serialize()* method

5

Deserialization: *BinaryFormatter*

```
Circle c;
Rectangle r;

BinaryFormatter binFormattter = new BinaryFormatter();

using (Stream fStream =
        new FileStream("mydata.dat", FileMode.Open, FileAccess.Read))
  {
    c = (Circle)binFormattter.Deserialize(fStream);
    r = (Rectangle)binFormattter.Deserialize(fStream);
  }
```

create new binary formatter

- open a stream to read,
- deserialize (restore) objects from the stream using the *Deserialize*() method (casting *Deserialize()* result is necessary)

6

Serialization: *XmlSerializer*

```
[Serializable]
public class Point
{
    public double x, y;
    public Point() { x = y = 0; }
    public Point(double newX, double newY) { x = newX; y = newY; }
}

[Serializable]
public class Rectangle
{
    public Point LowerLeft, UpperRight;
    public Rectangle() : this(0, 0, 0, 0) { }
    public Rectangle(double x1, double y1, double x2, double y2)
    {
        LowerLeft = new Point(x1, y1);
        UpperRight = new Point(x2, y2);
    }
}
```

- due to the requirements/limitations of the XML serializer, classes must be customised:
  - non-public members cannot be serialised
  - classes must have public default constructors

7

Deserialization: *XmlSerializer*

```
Rectangle r = new Rectangle(1, 1, 5, 3);

XmlSerializer xmlFormattter = new XmlSerializer(typeof(Rectangle));

using (Stream fStream =
    new FileStream("mydata.xml", FileMode.Create, FileAccess.Write))
{
   xmlFormattter.Serialize(fStream, r);
}
```

create new *XML* formatter – serialised object type must provided

- open/create a stream to write
- serialize (store) objects to the stream using *Serialize()* method

8

Serialization: *XmlSerializer*

```
<?xml version="1.0"?>
<Rectangle xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <LowerLeft>
    <x>1</x>
    <y>1</y>
  </LowerLeft>
  <UpperRight>
    <x>5</x>
    <y>3</y>
  </UpperRight>
</Rectangle>
```

example of serialised object

9

Deserialization: *XmlSerializer*

```
Rectangle r;

XmlSerializer xmlFormattter = new XmlSerializer(typeof(Rectangle));

using (Stream fStream =
        new FileStream("mydata.xml", FileMode.Open, FileAccess.Read))
 {
    r = (Rectangle)xmlFormattter.Deserialize(fStream);
 }
```

create new *XML* formatter – serialised object type must provided

- open a stream to read,
- deserialize (restore) objects from the stream using the *Deserialize()* method (casting *Deserialize()* result is necessary)

10

.NET non-generic collections

- namespace *System.Collections* contains implementations of various data structures such as dynamic array, queue, stack ...
- all data structures are polymorphic - they can store elements of any type (descendants of *System.Object*),
- the advantage of this solution is flexibility - one implementation for any type of object,
- the disadvantages of this solution are:
  - poor performance, especially for value types (boxing/unboxing overhead),
  - susceptibility to errors, any type of element can be inserted into the data structure - it's easy to make accidental errors (testing/casting types is required to protect against the consequences of errors),

11

Array (array–list)

resizable array, basic features:
- insert elements – *Add ()* ,
- remove the first occurrence of an element – *Remove()* ,
- remove element at the given index – *RemoveAt ()*,
- access to elements using the indexer [],
- current number of elements: *Count*,
- searching, inverting ...

```
ArrayList myAL = new ArrayList();
myAL.Add("Hello");
myAL.Add("World");
myAL.Add("!");

Console.WriteLine("The number of elements:" + myAL.Count );

while (myAL.Count > 0)
{
  object o = myAL[myAL.Count - 1];
  myAL.RemoveAt(myAL.Count-1);
  Console.WriteLine(myAL.Count + ":" + o.GetType() + ":(" + o + ")");
}
```

```
The number of elements:3
2:System.String:(!)
1:System.String:(World)
0:System.String:(Hello)
```

```
myAL.Add("Hello");
myAL.Add(32);
myAL.Add(5.4);

while (myAL.Count > 0)
{
   object o = myAL[myAL.Count - 1];
   myAL.RemoveAt(myAL.Count - 1);
   Console.WriteLine(myAL.Count + ":" + o.GetType() + ":(" + o + ")");
}
```

```
2:System.Double:(5,4)
1:System.Int32:(32)
0:System.String:(Hello)
```


12

Stack

stack, basic features:
- add an element to the collection: *Push()*,
- remove the most recently added element: *Pop()*,
- current number of elements: *Count*

```
Stack myS = new Stack();
myS.Push("Hello");
myS.Push(32);
myS.Push(5.4);

Console.WriteLine("The number of elements:" + myS.Count );

while(myS.Count>0)
{
  object o = myS.Pop();
  Console.WriteLine(myS.Count + ":" + o.GetType() + ":(" + o + ")");
}
```

```
The number of elements:3
2:System.Double:(5,4)
1:System.Int32:(32)
0:System.String:(Hello)
```


13

FIFO queue

first-in-first-out queue, basic features:
- add an element to the back of the queue: *Enqueue()*,
- remove an element from the front of the queue: *Dequeue()*,
- current number of elements: *Count*

```
Queue myQ = new Queue();
myQ.Enqueue("Hello");
myQ.Enqueue(32);
myQ.Enqueue(5.4);

Console.WriteLine("The number of elements:" + myQ.Count );

while(myQ.Count>0)
{
  object o = myQ.Dequeue();
  Console.WriteLine(myQ.Count + ":" + o.GetType() + ":(" + o + ")");
}
```

```
The number of elements:3
2:System.String:(Hello)
1:System.Int32:(32)
0:System.Double:(5,4)
```


14

Generic programming

- generic programming is the second (next to inheritance) mechanism supporting code reuse,
- generic types support safer, more efficient and more understandable code,
- in C++, templates are similar to macros of the C language preprocessor - they are expanded during compilation (but not necessarily in place), macro arguments are not calculated before expansion, templates are considered safer but have numerous disadvantages (e.g. no hiding information/implementation),
- C# generics are improved, comparing to the concept of templates from C++, they are effective code factories, they do not require access implementation source,
- particularly useful in creating libraries in collection classes,
- the implementation of generic code is more difficult - the user does not expect any cumbersome restrictions or unclear error messages,

15

Generic method

generic method definition

```
class GenericLibrary
{
    static public void Swap<T>(ref T a, ref T b)
    {
        T Temp;
        Temp = a;
        a = b;
        b = Temp;
    }
}
```

definition of a method declaring generic type *T* parameter (any number of type parameters is allowed), the method does not have to be static (however a library function often is)

open type: `Swap<T>`

closed type: `Swap<int>`

```
int a = 1, b = 2;

Console.WriteLine("a:" + a + " b:" + b);

GenericLibrary.Swap<int>(ref a, ref b);

Console.WriteLine("a:" + a + " b:" + b);
```

instantiation and invocation of the method with type argument *int*

16

Generic class (#1/2)

generic class definition

```
class GenericStack<TStackItem>
{
  private TStackItem[] vector;
  private uint Top;
  public GenericStack(uint Capacity = 100)
  {
    vector = new TStackItem[Capacity];
  }
  public void Push(TStackItem s) { vector[Top++] = s; }
  public TStackItem Pop() { return vector[--Top]; }
}
```

definition of a class declaring a type parameter *TStackItem*

```
GenericStack<int> st0 = new GenericStack<int>(200);

for (int u = 0; u < 10; ++u) st0.Push(u);
for (int u = 0; u < 10; ++u)
   Console.WriteLine(st0.Pop());
```

instantiation of the class with type argument *int*

instantiated class usage

17

Generic class (#2/2)

generics can be used together with inheritance/polymorphism

```
GenericStack<object> st0 = new GenericStack<object>();

st0.Push("Insert texts"); // insert text
st0.Push("and numbers");  // insert text
st0.Push(20);  // box int constant (System.Int32)
st0.Push(10);  // box int constant (System.Int32)

Console.WriteLine("Pop, output texts");
Console.WriteLine("sum numbers and print");
int sum = 0;
for (uint u = 0; u < 4; u++)
{
  object o = st0.Pop();
  if (o is string) Console.WriteLine(o);
  else sum += (int)o;
}
Console.WriteLine("Sum = " + sum);
```

```
Pop, output texts
sum numbers and print
and numbers
Insert texts
Sum = 30
```

but performance (boxing/unboxing) and casting (types) issues return

18

Generic types – declaring limitations

where keyword allows placing restriction/requirements for types used in instantiation of the generic type

```
class MyClass<TC, TS>
    where TC : class
    where TS : struct
{
}
```

types used to instantiate *MyClass* class must be respectively:
- *TC* – a class
- *TS* – a struct(ure)

```
public class MyGenericClass<T> where T : IComparable, new()
{
}
```

type *T* used during instantiation must:
- implement given interface (*IComparable*),
- implement default (parameterless) accessible (non-private) constructor

19

Generic Delegates (#1/2)

`Func<TParameter, TOutput>`	function: input parameters + result
`Action<TParameter>`	procedure: only input parameters
`Predicate<in T>`	predicate: function returning *bool* value
`Converter<TInput, TOutput>`	converter: input objects to output objects, useful for converting collections
`Comparison<T>`	comparison: sorting/ordering data in collection

20

Generic Delegates (#2/2)

example of function instantiation

`Func<TParameter, TOutput>` function: input parameters + result

```
class Test
{
  public Func<int, int, string> tempFuncPointer;
  public string t(int a, int b) { return "Hello"; }
}
class Program
{
  static void Main(string[] args)
  {
     Test t1 = new Test();
     t1.tempFuncPointer = t1.t;
     Console.WriteLine(t1.tempFuncPointer(1, 1));
  }
}
```

delegate instantiation

- object creation
- delegate assignment
- method invocation

21