

Topics for today:

- Generics
- Constructors and Inheritance
- Properties
- Iterators

```
Stack s = new Stack();  
s.Push(19);  
s.Push(28);  
int x = (int) s.Pop();  
int y = (int) s.Pop();  
  
class Stack  
{  
    private object[] _itemsArray;  
    ...  
    public void Push(object item)  
    { ... }  
    public object Pop()  
    { ... }  
}
```

Boxing  
Unboxing  
Cast required

```
Stack s = new Stack();  
s.Push("hi there");  
s.Push(28);  
int x = (int) s.Pop();  
int y = (int) s.Pop();
```

This compiles but  
throws an exception  
at runtime.

```
class Stack  
{  
    private object[] _itemsArray;  
    ...  
    public void Push(object item)  
    { ... }  
    public object Pop()  
    { ... }  
}
```

```
IntStack s = new IntStack();  
s.Push(19);  
s.Push(28);  
int x = s.Pop();  
int y = s.Pop();
```

No boxing or casting, and  
s.Push("hi there") would  
not compile. But we need a  
separate class for every type.

```
class IntStack  
{  
    private int[] _itemsArray;  
    ...  
    public void Push(int item)  
    { ... }  
    public int Pop()  
    { ... }  
}
```

```
class Stack<T>  
{  
    private T[] _itemsArray;  
    ...  
    public void Push(T item)  
    { ... }  
    public T Pop()  
    { ... }  
}
```

```
Stack<int> s = new Stack<int>();  
s.Push(19);  
s.Push(28);  
int x = s.Pop();  
int y = s.Pop();
```

s.Push("hi there");  
and  
string str = s.Pop();  
will not compile.

```
class Stack<T>  
{  
    private T[] _itemsArray;  
    ...  
    public void Push(T item)  
    { ... }  
    public T Pop()  
    { ... }  
}
```

C# allows:

- Generic reference types (classes)
- Generic value types (structs)
- Generic interfaces
- Generic delegates
- Generic methods

Advantages:

- Type safety
- Cleaner code
- Better performance

Classes included in the Framework Class Library

Generic Collection Class	Non-Generic
List<T>	ArrayList
SortedList<TKey, TValue>	SortedList
Dictionary<TKey, TValue>	Hashtable
SortedListDictionary<TKey, TValue>	SortedList
Stack<T>	Stack
Queue<T>	Queue
LinkedList<T>	(none)

Classes included in Wintellect's Power Collection Library  
<http://Wintellect.com> (download and use free of charge)

- BigList<T>
- Bag<T>
- OrderedBag<T>
- Set<T>
- OrderedSet<T>
- Deque<T>
- OrderedDictionary<TKey, TValue>
- MultiDictionary<TKey, TValue>
- OrderedMultiDictionary<TKey, TValue>

```
class Stack<T>
{
    private T[] _itemsArray;

    public bool Contains(T t)
    {
        for( int i = 0; i < _index; i++)
        {
            if (_itemsArray[i] == t) return true;
        }
        return false;
    }
}
```

This will not compile, because the compiler does not know whether the type supplied will support ==

```
class Stack<T>
{
    private T[] _itemsArray;

    public bool Contains(T t)
    {
        for( int i = 0; i < _index; i++)
        {
            if (_itemsArray[i].CompareTo(t) == 0) return true;
        }
        return false;
    }
}

public interface IComparable
{
    int CompareTo(object obj);
}
```

This will not compile, because the compiler does not know whether the type supplied is derived from IComparable.

```
class Stack<T>
{
    private T[] _itemsArray;

    public bool Contains(T t)
    {
        for( int i = 0; i < _index; i++)
        {
            if (_itemsArray[i].CompareTo(t) == 0) return true;
        }
        return false;
    }
}

public interface IComparable
{
    int CompareTo(object obj);
}
```

Casting this to IComparable could cause a run-time exception if T does not support IComparable and would lead to boxing if T is a value type.

```
class Stack<T> where T : IComparable
{
    private T[] _itemsArray;

    public bool Contains(T t)
    {
        for( int i = 0; i < _index; i++)
        {
            if (_itemsArray[i].CompareTo(t) == 0) return true;
        }
        return false;
    }
}

public interface IComparable
{
    int CompareTo(object obj);
}
```

We will still box value types.

```
class Stack<T> where T : IComparable<T>
{
    private T[] _itemsArray;

    public bool Contains(T t)
    {
        for( int i = 0; i < _index; i++)
        {
            if (_itemsArray[i].CompareTo(t) == 0) return true;
        }
        return false;
    }
}

public interface IComparable<T>
{
    int CompareTo(T t);
}
```

### There are three kinds of constraints

**Derivation constraints**

```
public class MyList<K,T> : IEnumerable<T>
    where K : MyBaseClass, IComparable<K>
```

**Constructor constraint**

```
public MyClass<T> where T: new()
```

**Reference/value constraint**

```
public MyClass<T> where T: class
public MyClass<T> where T: struct
```

### References

**An Introduction to C# Generics**  
Juval Löwy  
IDesign  
[http://msdn2.microsoft.com/en-us/library/ms379564\(vs.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms379564(vs.80).aspx)

**Introducing Generics in the CLR**  
Jason Clark  
Wintellect  
<http://msdn.microsoft.com/msdnmag/issues/06/00/NET/default.aspx>

### Visual Studio Example

#### Constructors and Inheritance

### Method Hiding

```
class Super
{
    public Super() { . . . };

    public void DoSomething() (not virtual)
    {
        . . .
    }
}

class Sub : Super
{
    public Sub() { . . . };

    public new void DoSomething()
    {
        . . .
    }
}
```

This "hides" the method of the parent, making it inaccessible through the derived class. This applies to any class member. A compiler warning is given unless the **new** keyword is used.

Properties

```

class Square
{
    private double side;

    public Square(double side)
    {
        this.side = side;
    }

    public void Grow()
    {
        side *= 2;
    }

    public double Side
    {
        get {return side;}
    }
}

class SquareUser
{
    private Square sqr =
        new Square(4);
    private double size;
    // possibly calls
    // sqr.Grow()
    // ...
    size = sqr.Side;
}
    
```

Property

Properties

```

class Square
{
    private double side;
    private const double MAX_SIDE = 10.0;
    public Square(double side)
    {
        this.side = side;
    }

    public void Grow()
    {
        side *= 2;
    }

    public double Side
    {
        get {return side;}
        set
        {
            if (value <= MAX_SIDE)
                side = value;
            else
                side = MAX_SIDE;
        }
    }
}

class SquareUser
{
    private Square sqr =
        new Square(4);
    private double size;
    // possibly calls
    // sqr.Grow()
    // ...
    size = sqr.Side;
    sqr.Side = 12;
}
    
```

- Things to do if the value in the "set" method is inappropriate:
- Do nothing
  - Assign a default value
  - Do nothing but log the event
  - Throw an exception

```

class Square
{
    private double side;
    private const double MAX_SIDE = 10.0;
    public Square(double side)
    {
        this.side = side;
    }

    public void Grow()
    {
        side *= 2;
    }

    public double Area
    {
        get {return side * side;}
    }
}
    
```

The get method can also do some processing.

Arrays

Single dimension

```

int[] myInts = new int[20];
...
Console.WriteLine(myInts[i]);
    
```

Multidimension

```

string[,] myStrings = new string[5,6];
double[,] myDoubles = new double[3,8,5];
...
Console.WriteLine(myDoubles[i,j,k]);
    
```

Jagged

```

Point[][] myPolygons = new Point[3][];
myPolygons[0] = new Point[10];
myPolygons[1] = new Point[20];
myPolygons[2] = new Point[30];
...
for (int x = 0; x < myPolygons[1].Length; x++)
    Console.WriteLine(myPolygons[1][x]);
    
```

Arrays

```

Point[][] myPolygons = new Point[3][];
myPolygons[0] = new Point[10];
myPolygons[1] = new Point[20];
myPolygons[2] = new Point[30];

for (int x = 0; x < myPolygons[1].Length; x++)
    Console.WriteLine(myPolygons[1][x]);

foreach (Point p in myPolygons[1])
    Console.WriteLine(p);

foreach (Point[] ptArray in myPolygons)
    foreach (Point p in ptArray)
        Console.WriteLine(p);
    
```

Note: inside a foreach loop we have read only access to the array elements.

### Arrays

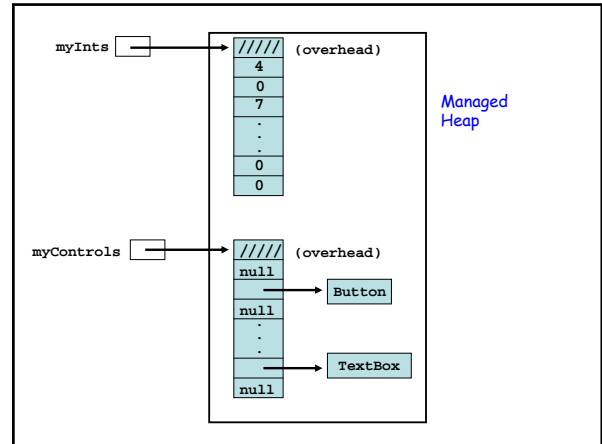
```
int[] myInts;      Creates a variable that can point to an array

myInts = new int[100]; Creates an array of 100 ints.
                        These ints are initialized to 0, and
                        stored, unboxed, in a memory block
                        on the managed heap.

Control[] myControls; Creates a variable that can point to an array

myControls = new Control[100];

Creates an array of Control references, initialized
to null. Since Control is a reference type, creating
the array creates references—the actual objects
are not created.
```



### Array Initialization

Single dimension

```
int[] myInts = new int[3] {1, 2, 3};
int[] myInts = new int[] {1, 2, 3};
int[] myInts = {1, 2, 3};
```

### Arrays are implicitly derived from System.Array

```
int[] myInts = new int[3] {1, 2, 3};
double[,] myDoubles = new double[3,8,5];
```

**Properties**

```
myInts.Length is 3
myInts.Rank is 1

myDoubles.Length is 120
myDoubles.Rank is 3
myDoubles.GetLength(1) is 8
```

**Static Methods**

```
Array.Sort(myInts);
Array.Sort(keys, items);
Array.Reverse(myInts);
Array.Clear(myInts);
int i = Array.IndexOf(myInts, 2);
```

**Iteration:** How does foreach work, and  
 How do we make it work with our classes?

```
class MyCollectionClass : IEnumerable
{ ... }

foreach (string s in myCollection)
{
    Console.WriteLine("String is {0}", s);
}

is transformed into

IEnumerator enumerator =
    ((IEnumerable) myCollection).GetEnumerator();

while (enumerator.MoveNext())
{
    string s = (string) enumerator.Current;
    Console.WriteLine("String is {0}", s);
}
```

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

```
class MyCollectionClass : IEnumerable
{
    CollectionPart[] _parts;
    ...
    ...
    public IEnumerator GetEnumerator()
    {
    }
}
```

```
class MyCollectionClass : IEnumerable
{
    CollectionPart[] _parts;
    ...
    ...
    public IEnumerator GetEnumerator()
    {
        return _parts.GetEnumerator();
    }
}
```

Since `_parts` is a `System.Array`,  
which is enumerable, it has a  
`GetEnumerator` method.

We can just return the `IEnumerator`  
that we get from calling that method.

- Suppose `MyCollectionClass` stores the parts in a custom linked list.
- It has a member `_head` that refers to the first node in the list.
- The nodes have `Data` and `Next` properties.

```
class MyCollectionClass : IEnumerable
{
    MySpecialLinkedList _parts;
    Node _head;
    ...
    public IEnumerator GetEnumerator()
    {
    }
}
```

In C# 2.0 we can implement the iterator with a `yield` block

```
class MyCollectionClass : IEnumerable
{
    MySpecialLinkedList _parts;
    Node _head;
    ...
    public IEnumerator GetEnumerator()
    {
        Node current = _head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

Visual Studio example using a generic class