Classes

C# has single inheritance

```
[access modifiers] class identifier [ : base class ]
{ class body }
```

access modifiers {public, private, protected, internal, protected internal}

- public, private, protected same as Java and C++
- internal accessible to methods in class's assembly
  assemblies are *.dll, or *.exe files w/ class
- protected internal protected OR internal access.
- private is default

Usually: variables are private and methods are public.

Methods and Properties of a class can be either instance members or static (class) members.

- Static members are accessed using the class name (shared).
- Static methods cannot directly access nonstatic members.
Constructors

Create instance objects of classes with new, invokes constructor

Constructors are similar to Java and C++
usually public, have no type, same identifier as class

Default constructor -- none written

If constructor is written and default constructor (one w/o arguments) is desired it must be written.

Static constructor will run before any instance of class is created.
to set static members. e.g. static constructorName() { ... }
equivalent private static string variable = value;

Private constructors (w/ no arguments, no body, and no public constructors) used with class having static data members can be used to hold "global" static constants and public methods.

Public constructor and public readonly data members allow run time setable constants.
Methods

primitive data members in methods have default values:

- numeric: `0`
- bool: `false`
- char: `'\0'`
- enum: `0`
- reference: `null`

**this** keyword similar to Java, equivalent to Smalltalk's self

every method has an implicit this pointer.

Method overloading is similar to C++ and Java.

methods have same type and identifier but different signatures
argument list vary by number, type or both.

Variable number of arguments: **params**

compiler constructs an array to match function signature

```csharp
access type methodName (params type[] args) {...}
...
methodName (arg1, arg2, arg3, ... argN);
```
Passing value arguments

Value types are passed into methods by value.

Passing value types by reference

// declare the method
[access] [type] identifier ( ref int a, ref double b){...} 
...
// invoke the method
[object] identifier (ref var1, ref var2);

C# compiler requires that var1 and var2 must have values.

Passing uninitialized value types by reference using keyword out

// declare the method
[access] [type] identifier ( out int a, out double b){...} 
...
// invoke the method
[object] identifier (out var1, out var2);
Property concept

OO concept of encapsulation: private data public accessor mutator methods.

+ object (data) representation is independent of client's (caller's) use. Change representation w/o changing client how it is used.

- Client can't directly operate on object directly. Client needs to know public interface of object and some idea of its concept (what is used for)

Properties are "lite, nested classes" that provide accessor / mutators methods.

  + data object is hidden
  + data usage appears to be direct.

```csharp
anObject.Property = aValue;

aValue = anObject.Property;
```
Property declaration

public class ClassIdentifier {
    private type propertyVariable;
    ....
    // property definition
    public type PropertyIdentifier {
        get {
            // statements to return value
            return propertyVariable;
        }
        set {
            // statements to set value
            propertyVariable = value;
        }
    }
    ....
}

ClassIdentifier aClass = new ClassIdentifier();
...
type thisValue = aValue;
aClass.PropertyIdentifier = thisValue;

keyword value is an implicit argument, it is the value used in the assignment (thisValue).
class AClass {
    private int anIntProperty;
    int min, max; // set with AClass constructor
    public int AnIntProperty {
        set {
            if (value >= min && value <= max)
                anIntProperty = value;
            else
                throw new ArgumentOutOfRangeException("anIntProperty");
        }
        get { return anIntProperty; } }
    // ... assume AClass is constructed ...
    try { AnIntProperty = someIntValue; } 
    catch (Exception e) {
        Console.WriteLine(e);
    }
}
PropertyDemo.cs file is posted off class page (C# examples).
<< read file in lecture>>

In the earlier Hello432.cs example used these Button properties:
Width, Height, Dock, Text, Click, BackColor

Properties can encapsulate primitive data, objects, events...
inheritance

Derived classes (subclasses) can be derived (extended) from base classes (superclasses)

Derived classes inherit all data members and member methods except constructors.

Base constructors can be called from a subclass's constructor using : base

```csharp
access class BaseClassName {
    public BaseClassName ( [ [type1] [arg1] ]* ) { ... }
}
```

```csharp
access class DerivedClassName : BaseClassName {
    public DerivedClassName ( [ [type2] [arg2] ]* )
        : base ( [ [type1] [arg1] ]* ) { ... }
}
```

<< compile, run inheritDemo.cs file >>
<< ViewPoly.cs version of ViewPoly.java wrt next slide >>
polymorphism & virtual methods

Virtual methods are used to define methods that will be override and used called polymorphically (in the context of the virtual method).

Virtual specifies the root of the inheritance graph for polymorphic method calls.

In base class

```csharp
access virtual type polyMethod(...) {...}
```

In derived class

```csharp
access override type polyMethod(...) {...}
```

C# adds the use of `new` with virtual methods to indicate that a virtual method with the same name has been introduced lower in the inheritance graph. (otherwise compiler generates a warning: `new || override`)

In later derived class

```csharp
access new virtual type polyMethod(...) {...}
```

The new virtual method hides (shadows) the initial virtual method (higher in the inheritance graph)
When a non-virtual superclass's method is overridden in a derived class C# compiler will issue a warning.

`new` can be used to avoid warning (not override).

To call a superclass's method from the subclass context use

```csharp
base.methodName(...);
```

Posted example program methods.cs and a listing of its output methods.out demonstrates the use of `virtual`, `override`, `new virtual`, `new`, and `base` constructor calls across 6 classes (Aclass .. Fclass). Each constructor calls a `show()` method to trace values.
Abstract methods define a contract between the API for a method (access, type, name, signature) and the definition of how it performs its behavior (body of method).

```csharp
access abstract type methodName(methodSignature);
```

Any class that has an abstract method must be an abstract class.

Abstract classes cannot be instantiated.

Abstract methods must be defined by subclasses of the abstract class.

An abstract method is also a "pure" virtual method.

Sealed classes cannot be subclassed.
Object, the root ...

All variables (value and abstract) derive implicitly from Object

Virtual methods of Object that are often overridden

- **Equals()** are 2 objects equivalent
- **GetHashCode()** used in collections
- **GetType()** access to the type object (debugging)
- **ToString()** formatter / output
- **Finalize()** memory mgmt for unmanaged resources

Boxing allows value types to be treated as reference types. (Boxed inside an Object).

Boxing is implicit with assignment to type **object**

Unboxing returns the value type

Unboxing must be explicit.

```csharp
int i = 123;
//Boxing is implicit
object o = i;

// unboxing must be explicit
int j = (int) o;
Console.WriteLine("j:{0}", j);
```
Classes can be nested inside each other.

The inner class has scope access to the out class

The inner class can be accessed using dot notation

```csharp
access class OuterClass {
  access type varName_i, varName_j;
  ...
  public OuterClass (type arg_i, type arg_j) { .... }
  ...
  access class InnerClass {
    access type innerMethod(...) {
      // can access varName_i, varName_j here ... }
    ...
    // in main ....
    OuterClass.InnerClass ic = new OuterClass.InnerClass();
    ...
  }
}
```
Structs are a value-type object-based abstract type.

    can't inherit from any type (implicitly inherits from object)
    can't be subclassed.
    can't have a default constructor -- one w/ no args

If new isn't called on a struct an instance w/ all fields zeroed is created.

.NET Foundation defines a richer Point struct than this example....

```csharp
struct Point {
    public int x, y;

    public Point (int X, int Y) {
        x = X; y = Y; }

    public override string ToString() {
        return(String.Format("{0}, {1}" , x, y)); }

}

... Point pt = new Point(100, 100);
```