Some C# differences from C++ and Java

"using" for namespaces instead of "import" for packages
namespaces are compiled classes stored in DLLs (base classes)

"Main" is uppercase "bool" instead of "boolean"

Console.WriteLine("...") not System.out.println("....")
can have argument substitution and formatting like C printf

    Console.WriteLine("iArray[{0}] == {1}", i, iArray[i]);

The arguments are substituted by index {0, 1} thus an index can
be reused in the format specification string.

using System;
class WriteLine
{
    public static void Main()
    {
        int i = 20;
        System.Console.WriteLine("show arg 3 times:
        \t first = {0} \t second = {0} \t third = {0}"", i);
    }
}

show arg 3 times:
    first = 20    second = 20    third = 20
C# overview

C# language incorporates, builds upon, and extends concepts in C++ and Java.

Most types, arithmetic operators, and structured statements are the same.

```
char, short, int, long, double
+ - * / %
```

Implicitly casts from a simpler type to a more complex (no loss of info).
Must explicitly cast from complex to simpler type (possible loss).

```
float f;
int i, j = 20;
...
f = j;
i = (int) f;
```

Value types (built in, primitive) stored on run time stack (local, args)
Reference types (objects) are allocated (new) on the heap.
C# has garbage collection.
Constants, enums

const type identifier = value;
const int dimensions = 3;

Once declared and initialized value cannot be changed.

Enumerations: typed named constants

enum identifier [:base-type] { enumeration list };
enum grades { f = 0, d = 1, c = 2, b = 3, a = 4 };

base-types must be of integral types
  { ushort, short, int, uint, long, ulong }
can't be char.
Safer if && switch

```c#
int count = 2;
if (count = 0) ...; // if (count == 0)....
```

A valid C++ conditional statement that assigns 0 to `count`.

C# if expression accept only boolean values and there is no automatic conversion of `bool` to `int` -- compile time error.

Switch does not have automatically fall through for all cases.

```c#
switch (switch expression) {
    case case1 :
        // valid
    case case2 :
        statement;  // compile error
    case case3 :
        statement;
        goto case4;  // valid, goto case1!
    case case4 :
        statement; break;
    case case5 :
        statement;
    [ default : statement; ]
}
```

Switch expression: integral type, char, enum, or `string` type
foreach

Easy iteration through a collection

```csharp
foreach (type identifier in expression) statement;
```

```csharp
using System;

class SeeArgs {
    static void Main(string [] args) {
        int i = 0;
        foreach (string token in args)
            Console.WriteLine("Arg {0} is {1}", i++, token);
    }
}
```

```
C:\> SeeArgs Something there is that doesn't love a wall,
Arg 0 is Something
Arg 1 is there
Arg 2 is is
Arg 3 is that
Arg 4 is doesn't
Arg 5 is love
Arg 6 is a
Arg 7 is wall,
```

C:\>
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 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

```csharp
// 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`

```csharp
// 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

```csharp
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;
```

**propertyVariable** is encapsulated by **PropertyIdentifier**

**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("anIntProptery");
        }
        get { return anIntProperty; } }
    // ... assume AClass is constructed ...
    try { AnIntProperty = someIntValue; }
    catch (Exception e) {
        Console.WriteLine(e); }
}
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] ]* ) { ... }
}

access class DerivedClassName : BaseClassName {
    public DerivedClassName ( [ [type2] [arg2] ]* ) : base ( [ [type1] [arg1] ]* ) { ... }
}
```
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(...);
```
abstract & sealed classes

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).

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

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

Abstract classes can not 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.

Managed DirectX 9.0 Light class is sealed -- API developers don’t want others subclassing Light in ways that wouldn’t work w/ rendering.
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);
```
nested classes

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);
```
Exceptions

Exception handling allows an application to respond to program states that are not attributable to errors in program construction or user input.

Exceptional situations "raise", "fire", or **throw** exceptions.

Un-handled, **caught**, exceptions cause program termination.

When an exception is thrown the CLR looks for a handler (code block) in the current method. If not found the search for a handler continues by popping the call stack until one is found or the default CLR termination handler is invoked.

When an exception is caught and handled execution continues from the handler -- it does not return to the where the exception was thrown! (Not really an interrupt).
try-catch-finally

Thrown exceptions are typed and pass an exception Object to the catch block.

Exception classes can be derived from existing exceptions.
   Exception property Message can be used to set a display string

Code that could cause an exception to be thrown is placed inside a try block.

Code that handles the exception is placed inside a catch block.

There can be many catch blocks.

Catch blocks should be sequenced in the order of priority. The first one encountered will be used.

Optional code guaranteed to execute regardless of an exception being handled or not, can be placed after a catch block in a finally block.
public aMethod () {
... try {
    // exception possible code
}
catch (exception1 e1) {
    // code to handle exception1
}
catch (exception2 e2) {
    // code to handle exception2
}
... catch (exceptionN eN) {
    // code to handle exceptionN
}
[ finally {
    // code to execute regardless of exception
} ]
}

<< see example catchException.cs >>
.NET event handlers

In .NET Framework events are implemented as delegate properties of GUI controls.

The event handler methods associated with events by convention have the following form:

```csharp
access void eventHandlerName
    ( object sender, EventArgs eventArg ) {
        ...
    }
```

The second argument can be any object derived from EventArgs.

Event handlers are "appended to" the delegate collection with:

```csharp
GuiControlObject.eventProperty +=
    new EventHandler(eventHandlerName);
```

for example

```csharp
cancelButton.Click += new EventHandler(doCancelEvent);
```
Interfaces are closely related to abstract classes where all members are abstract.

A class that implements an interface is required to define all the methods, properties, events, indexers of the interface.

An abstract class is part of a single inheritance subgraph. "is-a" relation.

A car is a vehicle.

An interface can be "mixed in" with many inheritance subgraphs. "implements a", “is-a” relation.

A car implements boughtWithCredit capability
A house also implements a boughtWithCredit capability.

Interfaces are implicitly public.
implementing interfaces

To define the interface ADT

```csharp
[access modifiers] interface [I]identifier [:base list]
   { interface method signatures * }

interface IScalable {...}   // implicitly public
```

Stylistically, interfaces are usually named starting with a 'I' to distinguish them from classes.

base list contains the interface this interface extends

all interface methods are implicitly public -- error if you set access

To implement the interface

```csharp
[access modifiers] class identifier
   [: [base class] [, interface]* ]
   { [declarations]*, [methods]*
     [interface method definition]1+ }

public class TextObject: DiagramObject, IScalable {...}
```
calling implemented methods

In the simplest case implemented methods can be called like other methods.

If the methods are invoked from a base class context that does not implement the methods than the object must first be tested to see if the interface is implemented and then cast.

```csharp
if (anObject is anInterface)
{
    anInterface thisInterface = (anInterface) anObject;
    thisInterface.aMethod(...);
}
```

The "as" operator will test for the implementation and cast if its implemented, otherwise it returns null.

```csharp
IScalable scalable = d as IScalable;
```

<< see IScalable1.cs example >>
Consider a diagram program composed of TextDiagram, VectorDiagram, and ImageDiagram objects that are subclasses of different classes. (eg, TextDiagram derives from Text – Text class defines editing behaviors). In the program you want to be able to scale a collection of these objects. << see IScalable2.cs example >>

... 

IScalable [] diagramObjects;  // assume assignment

... 

foreach (IScalable scalable in diagramObjects) { 
    scalable.scale(); }
**multiple interfaces**

C# support single inheritance and multiple implementation.

Interfaces can derive (subclass), and combine, other interfaces.

With multiple interfaces implementations there can be method name collisions, for example implementing methods from different vendors, or an implemented method and method defined in the implementing class.

Explicitly invoking an implemented method

```csharp
... InterfaceName1.methodName(...);
IFoo.compress();
...
InterfaceName2.methodName(...);
IBar.compress();
```

With method name collision, all but one of the methods must be explicitly invoke.
Explicit implementation

Implicitly implemented methods can be abstract or virtual and overridden or new in derived classes.

The interface's implementation can be hidden from a client by using explicit implementation. With explicit implementation the method cannot be abstract or virtual.

```csharp
interface IInterface {
    int P {get; set;} } // P is a property

interface IDerived: IInterface {
    new int P(); } // P is a method

class aClass : IDerived {
    int IInterface.P { get {...} } // explicit implement
    public int P() {...} } // implicit
```

Property P above is hidden (shadowed) by method P.
Abstract Vs Interfaces

Abstract
"is - a" relationship exists
There can only be one
Design needs to have variable member fields.
Noun + verb

Interface
"can-be-done" relationship exists
“is – a” relationship also exists (w/o data or defined behaviors).
There can be many different ADTs with this capability
verb – interfaces don't have member fields
no versioning (new)

Given single inheritance, interface is probably the starting design point for any abstract DT (data type) your considering .

Ask: "Are the defining ADTs of this abstract DT defining a "capability of" or a "member-of" the DT?"
Arrays

Arrays type System.Array
Collection class with easy to use C-style syntax

example methods properties  << see on-line doc>>
BinarySearch(), Clear(), Copy(), Sort(), Reverse(),
Length, Rank, Initialize(), SetValue()

type[] arrayName;
...
arrayName = new type [size];

or

type[] arrayName = new type [size];

or

type[] arrayName = new type [size] { v_1, v_2, ... v_size};
type[] arrayName = {v_1, v_2, ... v_size};

array values set to default value of type.
First element is at index 0 and last element is at Length - 1.
multi-dimensions

Regular Multi dimensional arrays

```c#
type[,] arrayName;
type[,] arrayName = new type [rows, columns];
type[,] arrayName = { {v_1, v_2, ...v_n}, ....,
                      {v_i, v_{i+1}, ...v_{i+n}};}

right most subscript moves the fastest (is nested)
```

Bounds checking is enforced.

```c#
int [,] vector = new int[3,4];
vector[2,0] is valid, vector[1,4] is an error.
```

Jagged arrays each dimension is a 1 dimension array

```c#
type[][] arrayName;
type[][] arrayName = new type [rows][];
arrayName[0][i] = new type[size_i];
arrayName[1][i+1] = new type[size_{i+1}];
...
arrayName[k][1] = aValue;  // assume type correctness
```

Note separate (C and C++ like) indexes "[]" with jagged arrays
Indexers

Indexers are properties that act like overloaded C++ \[ \] operator

Within a class syntactically treat its contents as an array.

```csharp
public type this [ type arg ] {  
  get { ... }  
  set { ... }  }
...

object[i] = aValue;     // using set and get
Console.WriteLine("object[{0}] = {1}", i, object[i]);
```

this refers to current object, 
get and set must be defined.

<< see indexer.cs textbook example and indexer2d.cs >>

The index values do not need to be integers, they can be defined wrt the application

<table>
<thead>
<tr>
<th>ListBoxTest</th>
</tr>
</thead>
<tbody>
<tr>
<td>- strings: string[]</td>
</tr>
<tr>
<td>- ctr: int</td>
</tr>
<tr>
<td>+ ListBoxTest(string[])</td>
</tr>
<tr>
<td>+ Add(String): void</td>
</tr>
<tr>
<td>+ GetNumEntries(): int</td>
</tr>
<tr>
<td>+ indexer: string property</td>
</tr>
</tbody>
</table>
Generic Collection interfaces

**IEnumerator<T>**  
iterate / enumerate through collection using foreach statement

```csharp
public interface IEnumerator<T> : IEnumerable
```

**ICollection<T>**  
provides CopyTo(), and Count, ISReadOnly, ISSynchronized and SyncRoot properties

```csharp
public interface ICollection<T> : IEnumerable<T>, IEnumerable
```

**IList<T>**  
used by array-indexable dynamic collections

```csharp
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
```

**IDictionary<K,V>**  
used by key / value based collections

**IComparer<T>**  
compares two objects in collection for sorting

```csharp
IDictionary<K,V>
```
List<T> class

List<T> is a generic, dynamic, "array"

```csharp
public class List<T> : IList<T>, ICollection<T>,
    IEnumerable<T>, IList, ICollection, IEnumerable
```

additional properties and methods << review on-line documentation >>

- **Capacity**
  - get / set number of elements List can hold
- **Count**
  - get / set number of elements in the List
- **Item**
  - get / set element at index – this is an indexer for `List` (error in text – not a method)
- **Add()**
  - adds object to list
- **AddRange()**
  - adds elements of ICollection at end of list
- **BinarySearch()**
- **Clear()**
  - removes all elements
- **Find()**
- **GetEnumerator()**
- **Sort()**
  - sort List
IComparable<T>

Sort(), or, compare items in List<T> using IComparable<T>

IComparable<T> requires implementation of the CompareTo() method.

```csharp
int CompareTo ( T other )
```

Many .NET classes implements IComparable with default case-sensitive implementation. (Int32, String ....)

```csharp
public sealed class String : IComparable, ICloneable, IConvertible, IComparable<string>, IEnumerable<string>, IConvertible, IEquatable<string>
```

Comparing is implemented by calling the CompareTo()

```csharp
tObject.CompareTo(tObject2)
```

returns:

-1 object < object2
0 object == object2
1 object > object2
Delegates are type safe "function pointers" specifying method to be invoked (called) at run time. Events are specialized (convenience) delegates.

Delegates are ADTs (class types) – declared and instantiated. Can’t be derived.

Delegates encapsulate a method, specific method signature and return type with a reference object.

```csharp
public delegate type_d delegateClassName ([type_i arg_i]+);
...
type_d aMethodName ([type_i arg_i]+) {
    ...
    return type_d; }
...
public delegateClassName delegateName (aMethodName );
...
delegateName ([type_i arg_i]+); // invokes aMethodName()
```

Instantiated delegates are objects that can be passed to methods…
delegate properties

Delegates have to be instantiated. The instantiation could be done “automatically” via the Class by defining the delegates as static.

To eliminate the instantiations of delegates that may not be used, the delegate instantiation can be replaced with properties (static or not). Now the delegate isn’t created unless it is requested (with a get).

```csharp
public static DelegateClassName delegateName =
    new delegateClassName(methodName);
```

is replaced with

```csharp
public static DelegateClassName DelegatePropertyName
    { get { return new DelegateName(methodName); } }
```
registering methods

Delegates can be multicast – more than 1 method is called when delegate is activated.

Methods can be registered with delegates when they are constructed and dynamically thereafter.

```csharp
aDelegate = method1 + method2;

aDelegate += method3;  // method1, method2 and method3
```

Methods can be unregistered

```csharp
aDelegate -= method2;  // method1 and method 3
```

Observer Pattern

The registering and unregistering of “handlers” with “event actions” is often referred to as publishing (event action) and subscribing (handlers) or the Observer Pattern. When the event observing object changes state its dependents (subscribers) are notified and updated automatically.