C#

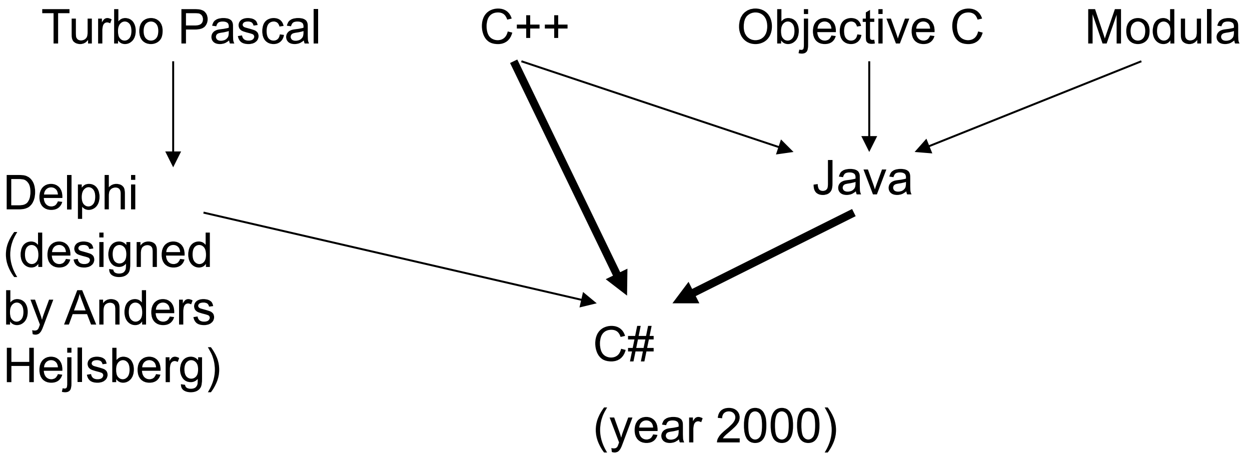
- Pitești University
- University of Bucharest
- Spiru Haret University

The Development Platform:
the Microsoft .NET Framework

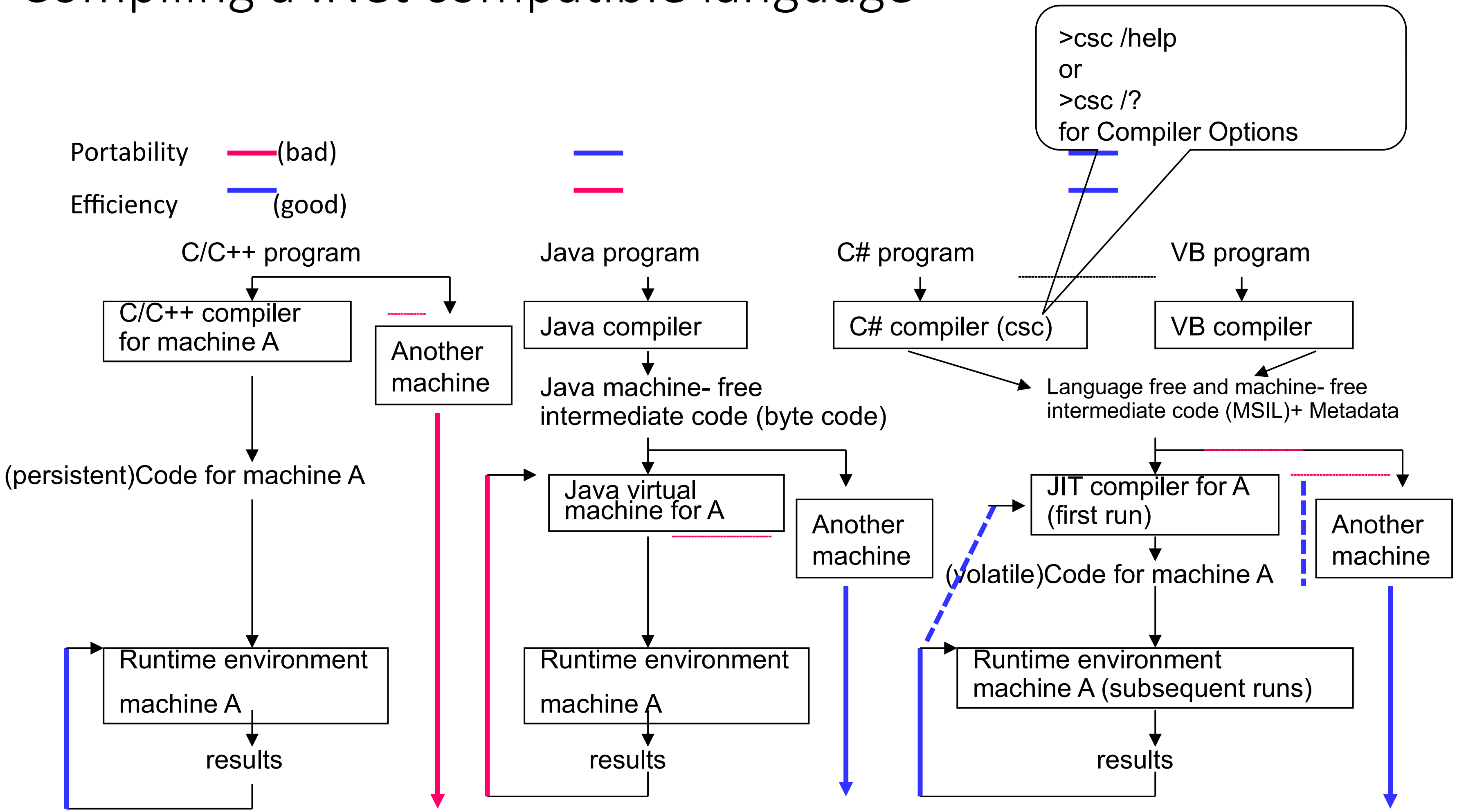
- The .NET Framework consists of two parts:
  - the common language runtime **(CLR)**
    - file loader,
    - garbage collector),
    - security system (code access security), and so on.
  - Framework Class Library **(FCL).**

History

- The principal inventors: Anders Hejlsberg, Scott Wiltamuth, Peter Golde
- ECMA Technical Committee 39 (TC39) Task Group 2(TG2) is responsible for the standardization of the language



Compiling a .Net compatible language



>csc /help
or
>csc /?
for Compiler Options

Portability (bad)

Efficiency (good)

C/C++ program

C/C++ compiler for machine A

Another machine

(persistent)Code for machine A

Runtime environment machine A

results

Java program

Java compiler

Java machine- free intermediate code (byte code)

Java virtual machine for A

Another machine

Runtime environment machine A

results

C# program

VB program

C# compiler (csc)

VB compiler

Language free and machine- free intermediate code (MSIL)+ Metadata

JIT compiler for A (first run)

Another machine

(volatile)Code for machine A

Runtime environment machine A (subsequent runs)

results

JIT= Just In Time compilation; MSIL=MicroSoft Intermediate Language

Below is a list of features C# and Java share, which are intended to improve on C++. [1]

- Compiles into *machine-independent language*-independent code which runs in a managed execution environment.
- *Garbage Collection* coupled with the *elimination of pointers* (in C# *restricted use* is permitted within code marked unsafe)
- Powerful *reflection* capabilities
- No header files, all code scoped to packages or assemblies, no problems declaring one class before another with *circular dependencies*
- Classes all descend from *Object* and must be *allocated on the heap* with new keyword
- *Thread support* by putting a lock on objects when entering code marked as locked/synchronized
- Interfaces, with *multiple-inheritance of interfaces*, *single inheritance of implementations*

Common Type System (4/4): predefined System.Object

- Here's another CTS rule. All types must (ultimately) inherit from a predefined type: **System.Object.**
- This **Object** is the root of all other types and therefore guarantees that every type instance has a minimum set of behaviors.
- Specifically, the **System.Object** type allows you to do the following:
  - Compare two instances for equality***.***
    (public virtual bool object.Equals(object obj))
  - Obtain a hash code for the instance***.***
    (public virtual int object.GetHashCode())
  - Query the true type of an instance***.***
    (public System.Type object.GetType()); it is ***not*** virtual
  - Perform a shallow (bitwise) copy of the instance.
  - Obtain a string representation of the instance's object's current state***.(string object.ToString())***

Example: System.Object (1/3)

```
// file 
using System;

    // The Point class is derived from System.Object.
    class Point
    {
        public int x, y;

        public Point(int x, int y)
        {
            this.x = x;
            this.y = y;
        }

        public override bool Equals(object obj)
        {
            // If this and obj do not refer to the same type, then they are not equal.
            if (obj.GetType() != this.GetType()) return false;

            // Return true if  x and y fields match.
            Point other = (Point)obj;
            return (this.x == other.x) && (this.y == other.y);
        }
```

Overriding Equals

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Equals
{
    class C
    {
        int x = 0;
    }
    class CO
    {
        C c;
        int x = 0;
        public CO(C y) { c = y; }
        public override bool Equals(object obj) {
            if (obj == null) return false;
            if (this.GetType() != obj.GetType()) return false;
            // now, the same type
            CO co = (CO)obj; // no exception
            // compare reference types
            if (!Object.Equals(c,co.c)) return false;
            if (!x.Equals(co.x))return false;
            return true;
        }
```

C# is more of a “pure” object-oriented language.

- Although it is based on C++, C# is more of a “pure” object-oriented language.
- One of the primary goals of C# is safety, so many of the problems that plague programmers in C and C++ are not repeated in C#.
- everything in C# is an object, even a C# program or an integer value.
- there is a single consistent syntax used everywhere
  - C++: direct or indirect representation (a pointer)
- the identifier you manipulate is actually a “reference” to an object.
  - a safe practice: always initialize a reference when you create it:
    `string s = "asdf";`
    or (better)
    `string s = new string("asdf”);`
  - //if not, possible error later: *use of unassigned local variable…*

```
10.ToString();
int i=0; i.ToString();
i.GetType().ToString();//System.Int32
i = new int(); i = 1;
```

Value types

- Unlike “pure” object-oriented languages such as Smalltalk, C# does not insist that every variable must be an object.
  - the allocation of many small objects can be notoriously costly.
- C# goes a step beyond Java; not only are values (rather than classes) used for basic numeric types, developers can create new value types in the form of enumerations (**enum**s) and structures (**struct**s).
  - new value types have all the advantages of both value types and objects.

Classes and inheritance

Classes and inheritance (C# MSDN Training, module 9)

- A C# class can extend at most one class.
    - Note: a struct does not support inheritance.

- A derived class inherits everything from its base class *except for the base class constructors and destructors.*
- A derived class cannot be more accessible than its base class.
  class Example{

  private class NestedBase { }

  public class NestedDerived: NestedBase { } // Error

  }

Looks like private inheritance in C++
C# has no private inheritance; all inheritance is public.

- Constructor declaration:
  To call a base class constructor from the derived class constructor, use the

keyword **base**.

  C(...): base(…) {...}

  C(…){…} means C(...): base() {...}

constructor initializer (:base)

- Constructor acces rules

class NonDerivable{

  private NonDerivable( ) { ... }

}

class Impossible: NonDerivable{

  public Impossible( ) { ... } // Compile-time error

}

there is no way for a derived class to call the base class constructor.

Scoping an identifier

- You can use the keywords **base and this** to also qualify the scope of an identifier. This can be useful, since a derived class is permitted to declare members that have

the same names as base class members.

```
class Token{
  protected string name;
}
class CommentToken: Token{
  int i=0;
  public void Method(string name, int i){
  base.name = name;
  this.i=i;
  }
}
```

- Unlike in C++, the name of the base class, such as **Token** in the example

is not used (Token::name).

- The keyword **base** unambiguously refers to the baseclass because in C# a class can extend one base class at most.

Protected members

- Methods of a derived class can only access their own inherited ***protected***

members.

  - They cannot access the protected members of the base class through references to the base class.

```
class Token{protected string name;}
class CommentToken: Token{
void AlsoFails(Token t){
    Console.WriteLine(name); //OK
Console.WriteLine(t.name); // Compile-time error
}
     }
```

  - Many coding guidelines recommend keeping ***all data private*** and using protected access only for methods.
  - protected access modifier cannot be used in a struct (structures does not support inheritance)

Versioning, explicit virtual chains, broken chains

Implementing methods

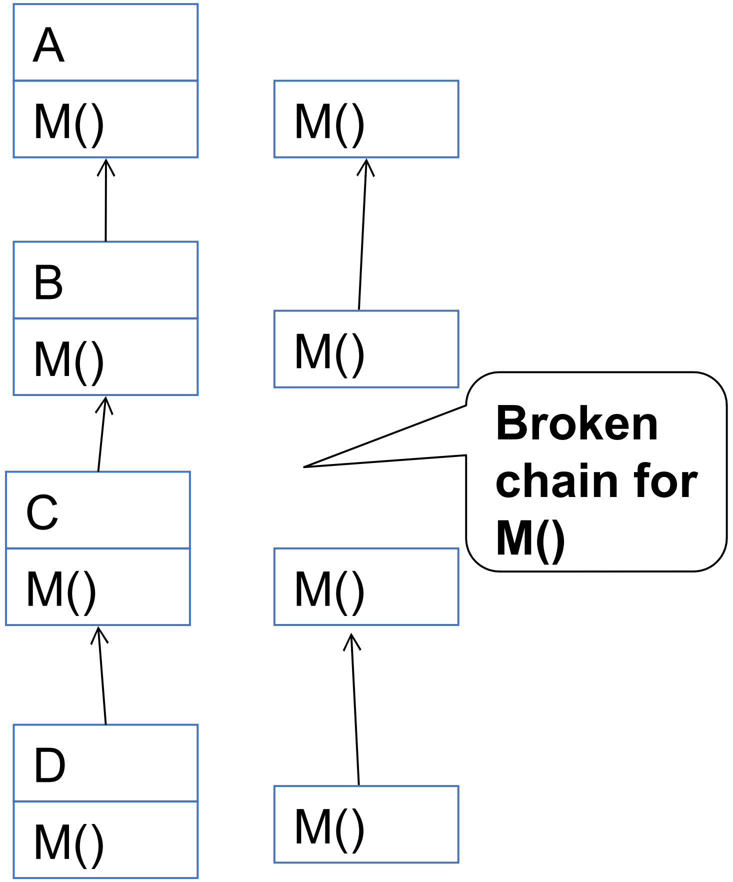
- You can redefine the methods of a base class in a derived class when the methods of the base class have been designed for *overriding.*
- A **virtual** method specifies an implementation of a method that can be polymorphically overridden in a derived class.
  - You cannot declare virtual methods as static (polymorphism works on objects, not on classes).
  - You cannot declare virtual methods as private (they cannot be polymorphically overridden in a derived class).
- An **override** method specifies another implementation (or ***version)*** of a **virtual** method.
  - You can only override identical inherited virtual methods.
  - You must match an override method with its associated virtual method
    - the same acces level
    - the same return type
    - the same signature
  - You can override an override method.
  - You cannot implicitly declare an override method as virtual.
  - You cannot declare an override method as static or private.

Using new to broke a chain (hide methods)

```
namespace MyVeryFirstProgram
{
  using System;

  class A
  {
    public virtual void M() { Console.Write("A"); }
  }
  class B : A
  {
    public override void M() { Console.Write("B"); }
  }
  class C : B
  {
    new public virtual void M() { Console.Write("C"); }
  }
  class D : C
  {
    public override void M() { Console.Write("D"); }
    static void Main()
    {
      D d = new D(); C c = d; B b = d; A a = d;
      d.M(); c.M(); b.M(); a.M();
    }
  }
}
```

The program displayes:
DDBB



Fragile Base Class Problem

Fragile base class problem

Changes in base class implementation, **causing a problem in the subclass**.

- Java example:

```
class BaseClass {
   public void display() {
      System.out.println("BaseClass.display()");
   }
   public void callMultipleTimes(int n) {
      for (int c1 = 0; c1 < n; c1++) {
         display();
      }
   }
}

public class Fragile{
     public static void main(String args[] ){
     BaseClass b= new BaseClass();
     b.callMultipleTimes(3);
     }
}
```

What if BaseClass changes?

An extension:

```
class DerivedClass extends  BaseClass {
   public void display() { //overriding
     counter++;      // new
      System.out.println("DerivedClass.display()");// modified
   }
}

   public int counter=0;
}
```

Changes, to count displays

```
public class FragileExtension{
     public static void main(String args[] ){
     DerivedClass b= new DerivedClass();
     b.callMultipleTimes(3);
     System.out.println(b.counter);
     b.callMultipleTimes(3);
     System.out.println(b.counter);
     }
}
```

FragileExtension displays 3 and 6 etc.

Base class changes causing problems

A new version of the base class

```
class BaseClass {
   public void display() {
      System.out.println("BaseClass.display()");
   }
   public void callMultipleTimes(int n) {
      for (int c1 = 0; c1 < n; c1++) {
      System.out.println("BaseClass.display()");
      }
   }
}
```

Change: no call to display()

The same FragileExtension .class (no need for compilation), with the new version of BaseClass.class displays 0 and 0 etc.! No counting!

Explanation: lack of the hierarchy contract

- How to forbid some changes? ??????????????????
- C# approach for fragile base class problem

Exercise: Spot the bugs

```
class Base
{
   public void Alpha() { ... }
   public virtual void Beta() { ... }
   public virtual void Gamma(int i) { ... }
   public virtual void Delta() { ... }
   private virtual void Epsilon() { ... }
}
class Derived : Base
{
   public override void Alpha() { ... }
   protected override void Beta() { ... }
   public override void Gamma(double d) { ... }
   public override int Delta() { ... }
}
```

Error: 'Base.Epsilon()': virtual or abstract members cannot be private

Error: 'Derived.Alpha()': cannot override inherited member 'Base.Alpha()' because it is not marked virtual, abstract, or override

Error: 'Derived.Beta()': cannot change access modifiers when overriding 'public' inherited member 'Base.Beta()'

Error: 'Derived.Gamma(double)': no suitable method found to override

Error: 'Derived.Delta()': return type must be 'void' to match overridden member 'Base.Delta()'