From C++ to C# (and C++ standard library to .NET)

Cpt S 321

Washington State University

String

- string vs String
  - String is a class declared in the System namespace in .NET.
  - string (lowercase s) type that maps to System.String.
- Strings in C# are immutable
- Once the object is instantiated, it cannot be changed in any way
- If the string is initialized as “ABCDE” then it will stay “ABCDE” in memory and cannot be modified.
- Methods: Substring (!), Replace (!), ToLower (!), **IndexOf**, **StartsWith**, and many others …

String (cont.)

Ex.1

```
string s = “Hello”;
string s1 = “_World!”;
s += s1;
Console.WriteLine(s); // Output ?
```

Ex.2

```
string s = "Hello ";
string s1 = s;
s += "World";
Console.WriteLine(s); // Output ?
System.Console.WriteLine(s1); // Output ?
```

Ex.3

```
string a = "hello";
string b = "h";
b += "ello";
Console.WriteLine(a == b);
                        // Output?
Console.WriteLine((object)a
        == (object)b); // Output?
```

StringBuilder

- StringBuilder class for mutable strings
- Better performance when your program has many string manipulations.
- Do not automatically replace all String by StringBuilder: String operations are highly optimized
- String versus StringBuilder, what to consider:
  - What is the number of changes you plan to make to a string? (Remember: String is immutable)
  - Will you need search methods? (The String class has convenient methods to search)
- StringBuilder - important properties: **Length**, **Capacity**, **MaxCapacity**
- Creating a StringBuilder:
  1. StringBuilder sb = new StringBuilder(); `// default capacity, i.e., 16 characters`
  2. StringBuilder sb = new StringBuilder("ABC", 50); `// explicitly setting the capacity`

A couple of things on *conversion*

```
1. int x = 12345;     // int is a 32-bit integer

2. long y = x;        // Implicit conversion to 64-bit integer

3. short z = x;       // Compilation error as int is 32-bits and we are
                      // trying to fit it to 16 => information loss

4. short z = (short)x;  // Explicit conversion to 16-bit integer.
                        // Required to solve the compilation error in 3.
```

A couple of reminders on *casting*

```
// Ex. 1:
1. Dog aDog = new Dog();
2. Animal anotherAnimal = aDog; // upcast, always succeeds
3. Console.WriteLine(anotherAnimal.Eat()); // Outputs: “Yummy!”
4. Console.WriteLine(aDog.Bark());         // Outputs: “Woof!!”
// Ex. 2:
5. Animal anAnimal = new Animal();
6. Cat aCat = (Cat)anAnimal; //explicit downcasting. If not
present, we get a compilation error. Use when sure that
the cast will succeed dynamically. In this example, we
will get a runtime error!
7. Console.WriteLine(aCat.Purr()); // compiles but will not
                                   // be reached at runtime
// Ex. 3: (alternative to Ex. 2 when not sure if a
//         cast will succeed dynamically:
8. Animal anAnimal = new Animal();
9. if (anAnimal is Cat anotherCat)
{
10.      Console.WriteLine(anotherCat.Purr());
}
```

```
public class Animal
{
        public String Eat() => "Yummy!";
}

public class Dog : Animal
{
        public String Eat() => "Yummy woof!";

        public String Bark() => "Woof!!";
}

public class Cat : Animal
{
        public String Purr() => "Purr!";
}
```

Boxing and Unboxing

- Boxing: converting value type instance to a reference type instance
- Unboxing: converting reference type instance to a value type instance
- Example:

```
1. int i = 123;     // we are declaring a value type instance
2. object o = i;    // we are "boxing" i meaning copying the value of i (from the stack)
                    // into an object o (on the heap)
                    // note that boxing is implicit
3. int j = (int)o;  // we are "unboxing" o - copying the value of o into an integer j
                    // note that boxing is explicit
```

- What is the value of o if we change the value of i?

```
4.   i = 456;
```

Answer: o is still 123 (remember that we are copying values!)

Arrays

- Arrays are OBJECTS in C#
- They have properties and methods unlike arrays in C++
- Length property tells you the size of the array
  - Is read-only and cannot be set (**why?**)
- Array indices are checked and if out of bounds, *exceptions* are thrown
- Examples:
  1. int[] anArray = new int[]{1,2,3,4,5};
  2. Console.WriteLine(anArray[5]); `// throws an exception`
  3. int[,] anotherArray = new int[3, 6]; `// a two dimensional array`

C#:List (C++:vector)

- Link to List class on MSDN
- Generic class (like a template class)
- System.Collections.Generic Namespace
- Holds a collection of objects of the same type
- Indexed access
- Can remove at any valid index (RemoveAt)
- C# list has **Count** property (equivalent to C++ vector's size() function)
- Examples:

```
1. List<int> myList = new List<int>();
2. myList.Add(42);
3. Console.WriteLine(myList[0]);
```

C#:Dictionary<TKey, TValue> (C++:unordered_map)

- Link to Dictionary class on MSDN
- Hash table implementation
  - Collection of key/value pairs
  - One key maps to one value
- Generic class → Can specify types for both the keys and values
- Has Count property that indicates the number of key value pairs in the collection
- Has Add method to add a new key/value pair
- operator[] allows you to access items by key

C#:Dictionary - Example

```
Dictionary<string, int> students = new Dictionary<string, int>();
students.Add(“Student A”, 12345678);
students.Add(“Student B”, 87654321);
Console.WriteLine(students[“Student A”]);
Console.WriteLine(students[“Student B”]);
// Output?
12345678
87654321
```

Q: Anything wrong with the design here?

C#:HashSet<T> (C++:unordered_set)

- Link to HashSet class on MSDN
- Hash set implementation
  - Collection of unique items (no duplicates)
  - Item insertion and lookup is close to O(1) (provided the hash table doesn't need to resize internally)
- Generic class → Can use it to store a set of ANY type of object
- Add function
  - Adds the item to the set if it isn't already there
  - Otherwise does nothing
- Count property: the number of elements that are contained in the set
- HashSet vs Dictionary

Stacks and Queues

- They’re in the standard C++ library and also in the .NET framework

- Within the System.Collections.Generic namespace:
  - Stack class
  - Queue class

Math

- There's a Math class (System.Math) that provides all the basic mathematical operations and values
- **Static** fields
  - Math.E
  - Math.PI
- **Static** methods
  - Math.Sin, Math.Cos
  - Math.Abs
  - Math.Floor, Math.Ceiling
  - many more methods

Instance (non-static) versus class (static) members

Employee.cs

```
public class Employee
{
    public string Id { get; set; }
    public string Name { get; set; }
    // Other fields, properties, methods
}
```

EmployeeManager.cs

```
public class EmployeeManager
{
    private static int employeeCounter=0;

    public static int EmployeeCounter
    {
        get => employeeCounter;
    }

    public static int IncrementEmployeeCounter() =>
        ++employeeCounter;
}
```

Program.cs

```
public static void Main(string[] args)
 {
      Employee e1 = new Employee();
      e1.Name="Venera Arnaoudova";
      Employee e2 = new Employee();
      e2.Name="John Smith";
      WriteLine(e1.Name== e2.Name);
}
```

```
// Output:
False
```

Program.cs

```
public static void Main(string[] args)
   {
       for(int i=0;i<4;i++)
       {
           EmployeeManager.IncrementEmployeeCounter();
           WriteLine(EmployeeManager.EmployeeCounter);
       }
   }
```

```
// Output:
1
2
3
4
```

No more limits.h

- In C++ you had various min/max values for several types defined in limits.h

- In C# such limits are available as static fields from the types themselves
  - int.MinValue, int.MaxValue
  - Similar things exist for char, byte, short, ushort, uint, float and double

- What can we say about the **cohesion** of the types then?
- What can we say about the **coupling** of the framework?

Cohesion

- Cohesion: for a class, cohesion measures how closely related fields, properties, and methods of that class are.
- We strive for **HIGH** cohesion

LOW cohesion:

```
public class Car
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }
    public int NumberOfEmployees { get; set; }
    public float EmployeeSalary { get; set; }
}
```

HIGH cohesion:

```
public class Employee
{
    public float EmployeeSalary { get; set; }
}
```

```
public class CarFactory
{
    public int NumberOfEmployees { get; set; }
}
```

```
public class Car
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }
}
```

Coupling

- Coupling: measures how interconnected entities (ex. classes) are
- We strive for **LOW** coupling

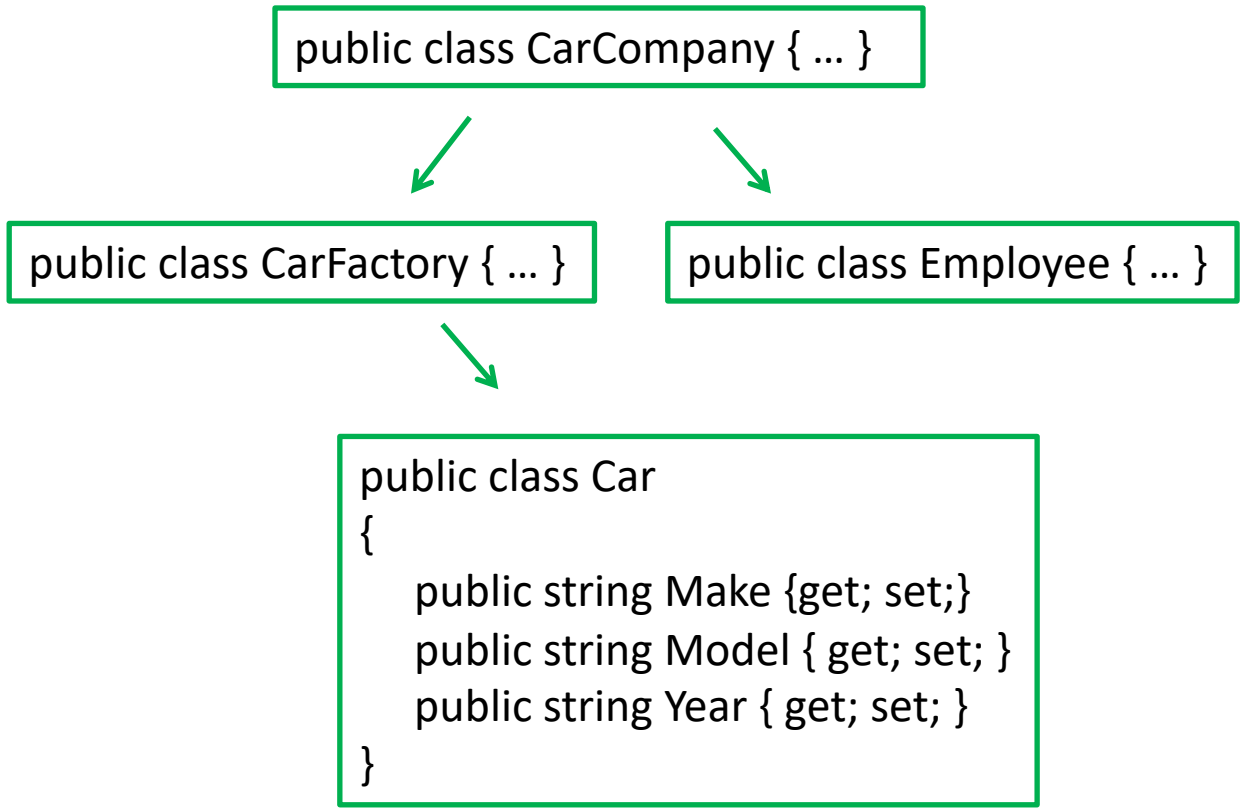
HIGH coupling:

```
public class Car
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }
    public int NumberOfEmployees { get; set; }
    public float EmployeeSalary { get; set; }
}
```

public class Employee { … }

public class CarFactory { … }

LOW coupling:



Use MSDN (Microsoft Developer Network)

- More information found on MSDN
- Use the links within these notes as starting points and get a feel for how to search for information about types in the .NET framework
- Web searches like “string class MSDN” should get you easy access to information about types in C#/.NET

Let’s touch base!

- Any questions/doubts/comments?

Let’s create the skeletal code for our main program (Program.cs)

- We want our program to behave as shown below:

```
C# Demos
 1 = Show the pass by reference and pass by value scenario
 2 = Show Hello World on the screen
 3 = Write Hello World in a file
 4 = Adding two numbers in a linked list
 0 = Quit
```

  - Print a menu for the user and ask them to choose
  - Parse the user input and allow them to choose what to do next

Hello World - TODOs for today

1. Implement Angle as a class and as a structure (use the lecture notes “IntroToCSharp” p.14 as a starting point)
   - Show a scenario where you observe that structures are passed by value and classes by reference

2. Create a BasicMessageClass
   - Add a field “message”
   - Add a property ”Message”
   - Add 2 constructors (default and with 1 parameter – the message)
   - Add a ”ShowMessage” method

Hello World (cont.)

3. In the main program
   - Create an instance of BasicMessageClass with ”Hello World!”
   - Show the message on the console (option 2)
   - Show the message in a text file (option 3)

```
using (System.IO.StreamWriter sWriter =
            new System.IO.StreamWriter(“myFirstFile.txt”))
{
    // your code goes here
}
```

   - Show the message in a …

4. Implement a linked list to store positive integers
   - Create a class LinkedListNode
   - In LinkedListNode we need a constructor
   - Create a class LinkedList
   - Implement method Add

Where to start?

- Open VS/Rider
- Create a new solution if you haven't done it already. **Use this solution for all in-class exercises**. You can call it 'CptS321-in-class-exercises' for example.
- Create a new project. Mine is called '**HelloWorld**' because this is our first C# project. This project can be a Console Application.
- VS/Rider automatically create a class for you called Program. This will be the main program that will display the menu and ask the user for input. Check the next slide for a template.

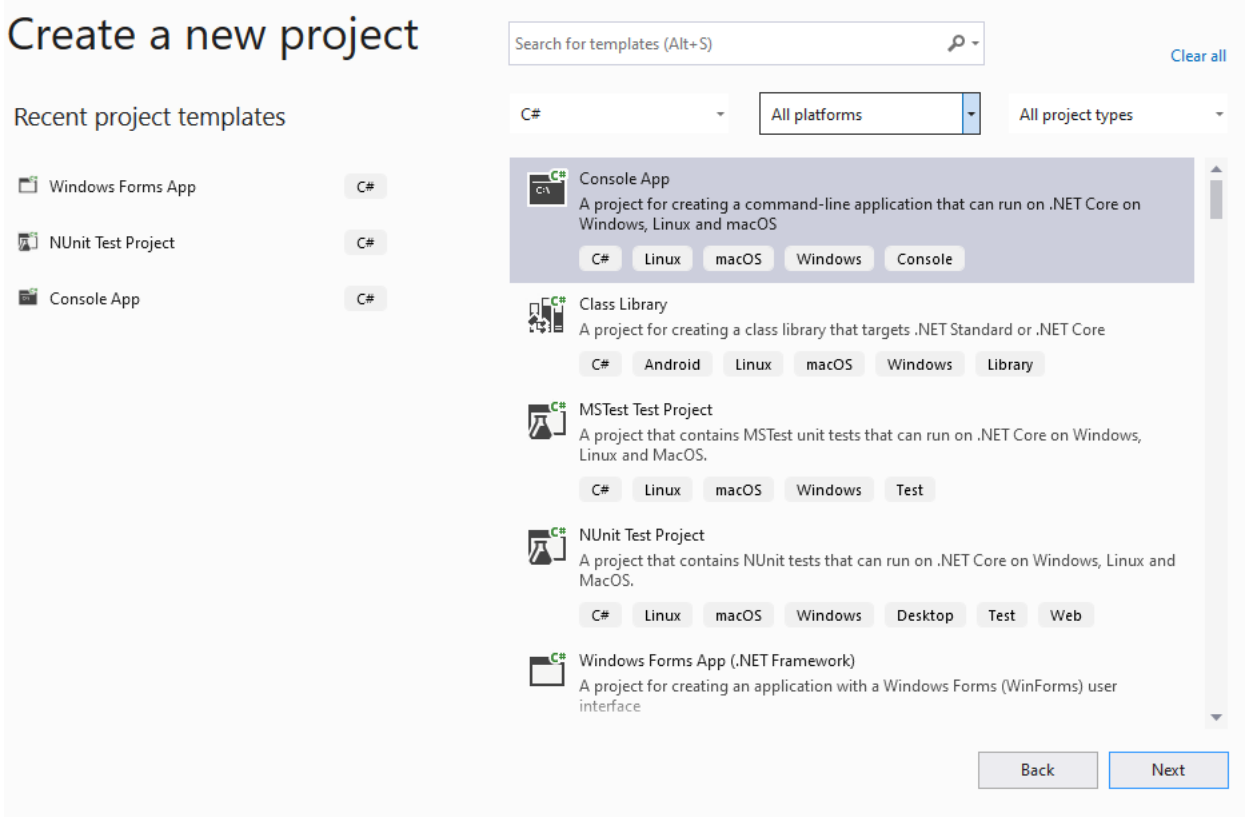
Where to start – a typical template for our main program

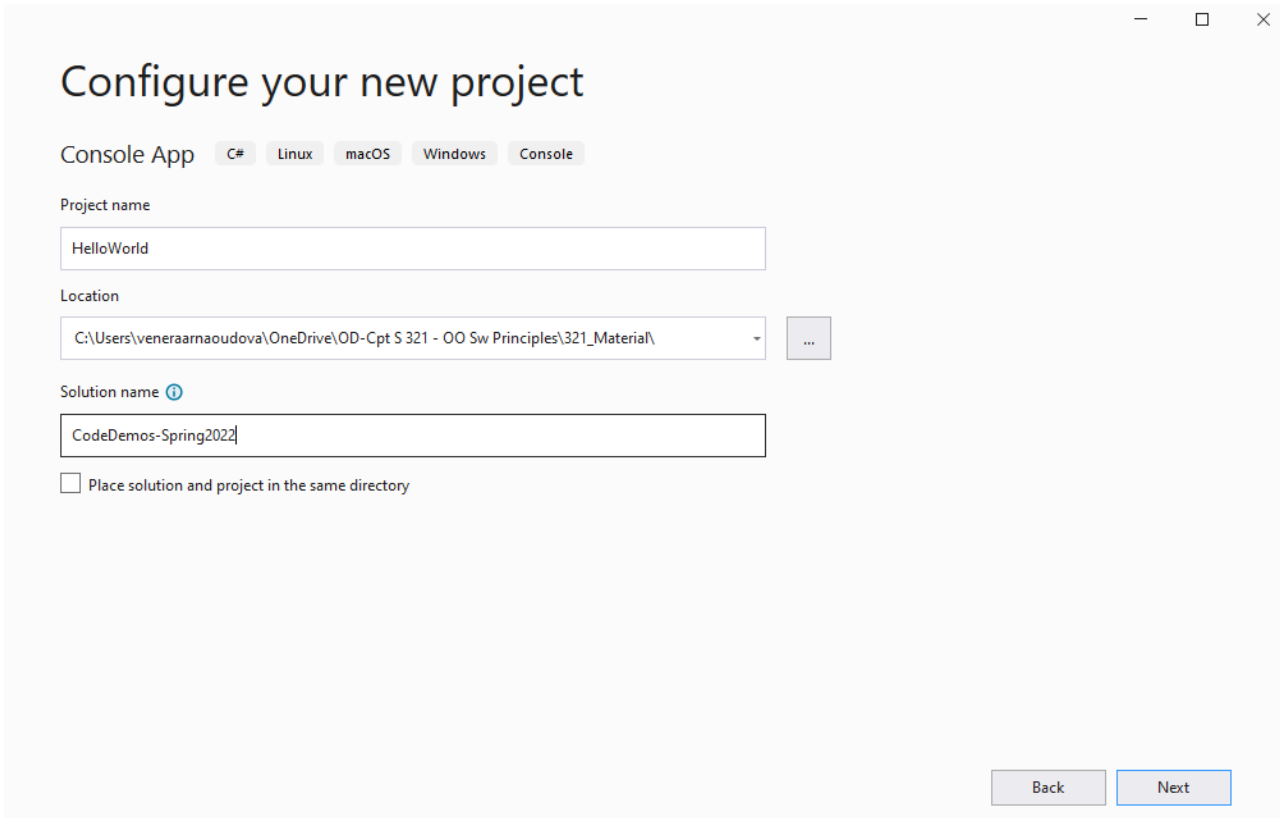
```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Your code goes here...");
    }
}
```

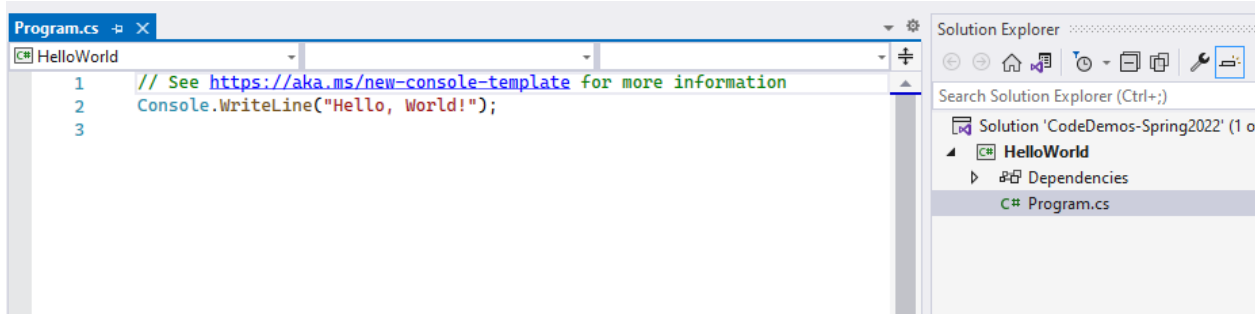
Where to start – C#9 and forward: “top-level statements”

```
Console.WriteLine("Your code goes here...");
```

A few screenshots that might help (VS)







A few screenshots that might help (Rider)