CSHARP-3

DATA TYPES AND DECLARATIONS

OUTLINE

- 1. Program Structure Review
- Lab: Create a project; create the 2nd Project, Write codes, debug
- 2. Types
- 3. Value Types
- 4. Reference Types
- 5. Variables

PROGRAM STRUCTURE

- The key organizational concepts in C# are ***programs***, ***namespaces***, ***types***, ***members***, and ***assemblies***.
- C# **programs** consist of one or more source files.
  - Programs declare **types**, which contain **members** and can be organized into **namespaces**.
    - **Classes** and **interfaces** are examples of **types**.
    - **Fields**, **methods**, **properties**, and **events** are examples of **members**.
- When C# programs are compiled, they are physically packaged into assemblies. Assemblies typically have the file extension `.exe` or `.dll`, depending on whether they implement ***applications*** or ***libraries***, respectively.

How?

C# PROGRAMS

Source file(s)

Namespaces

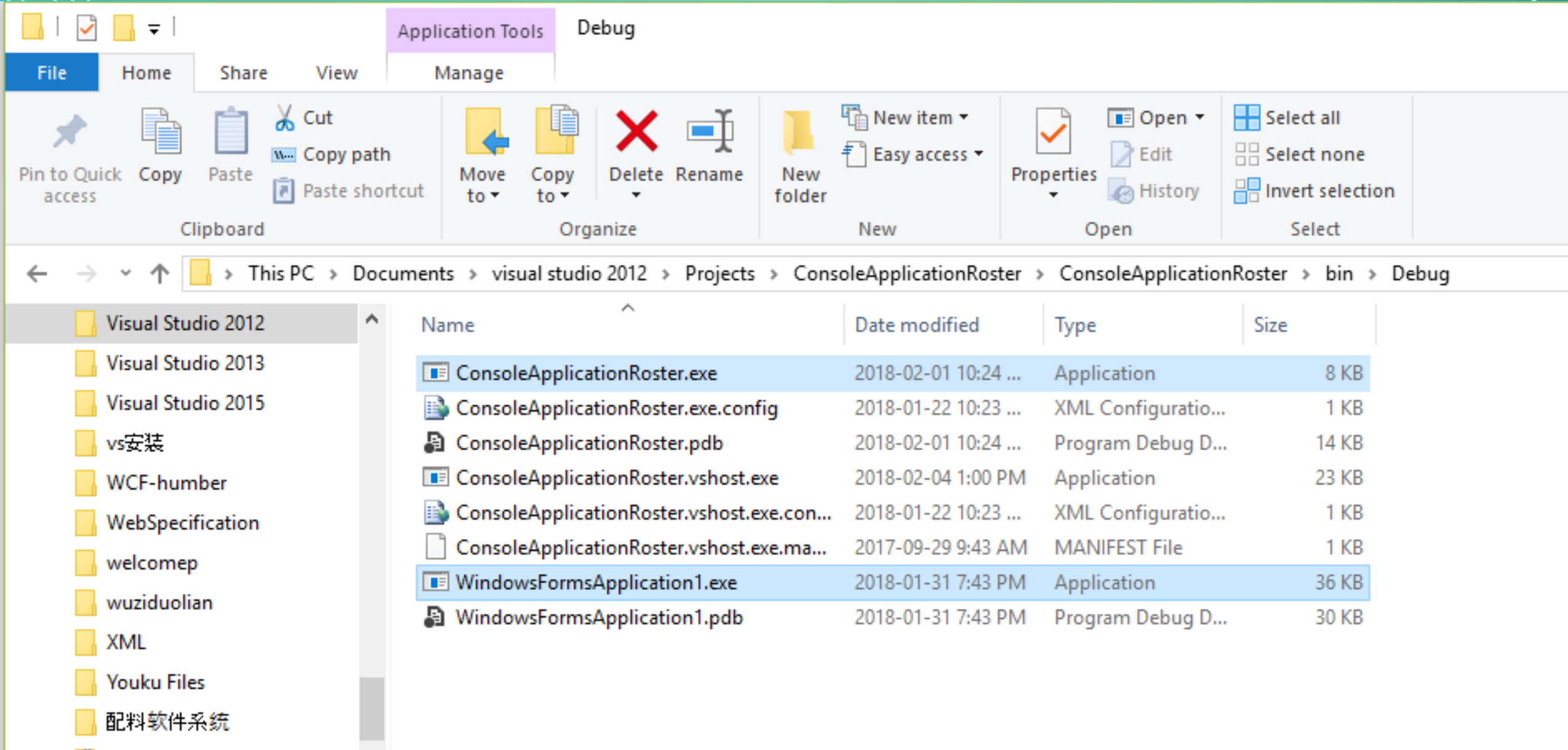
Types (eg. Classes and interfaces)

Members:
Fields, methods, properties, and events

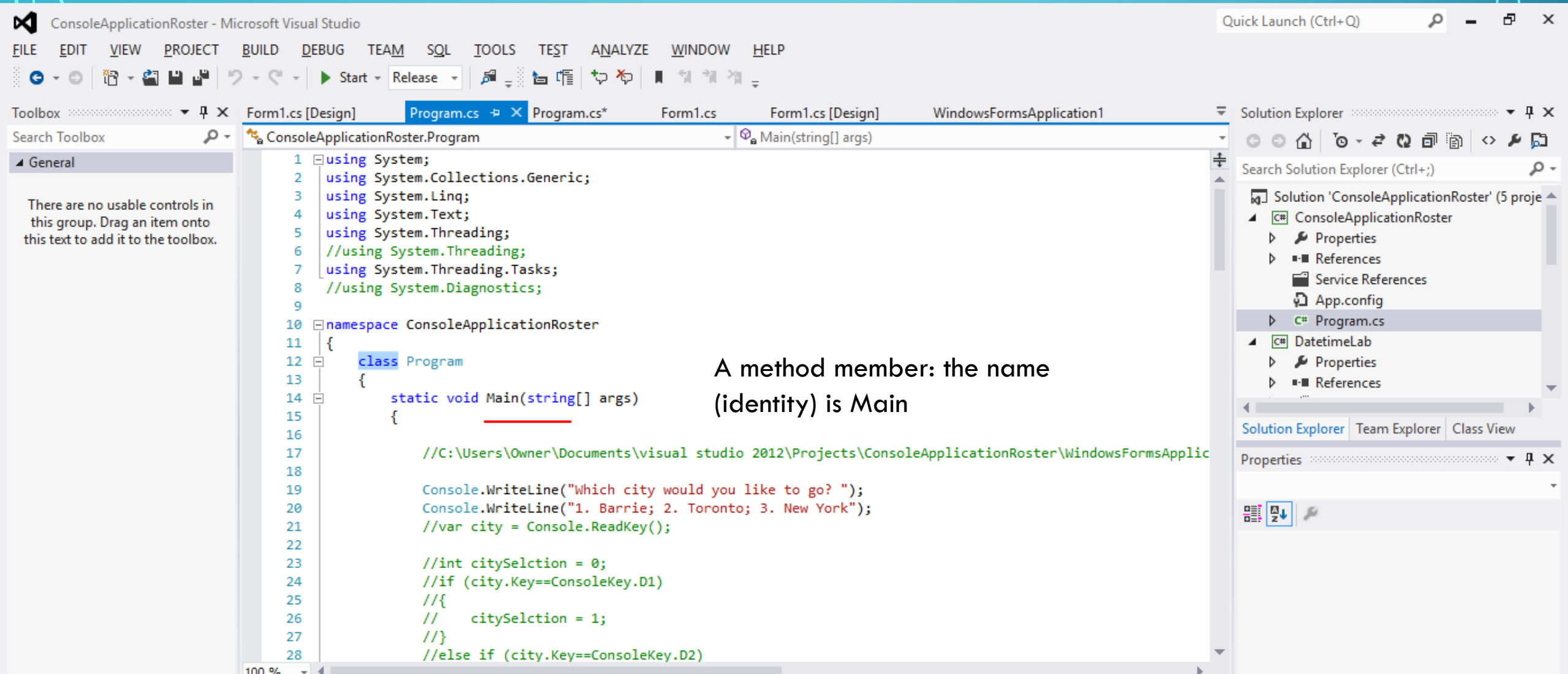
Compiled

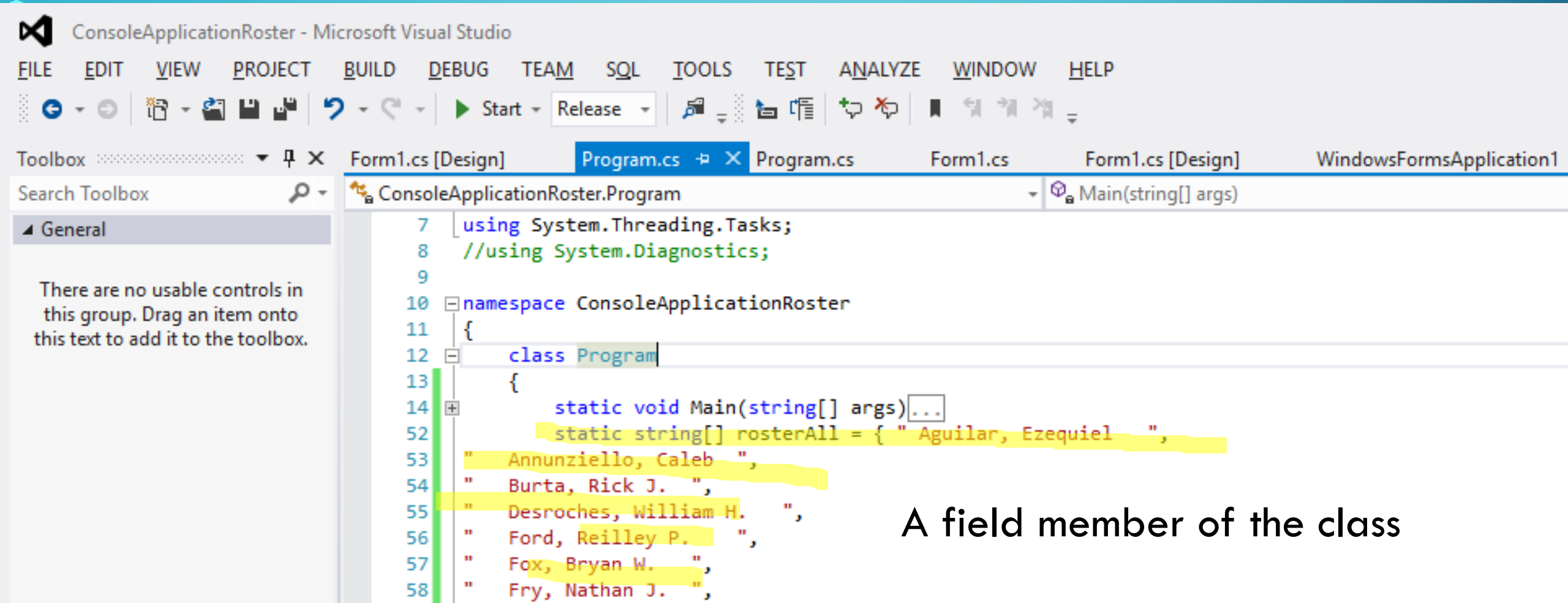
Assemblies (dll or exe files)

ASSEMBLIES – DELIVERABLE



SOURCE FILES





A field member of the class

LAB

- Review Assignment 2: Step 1

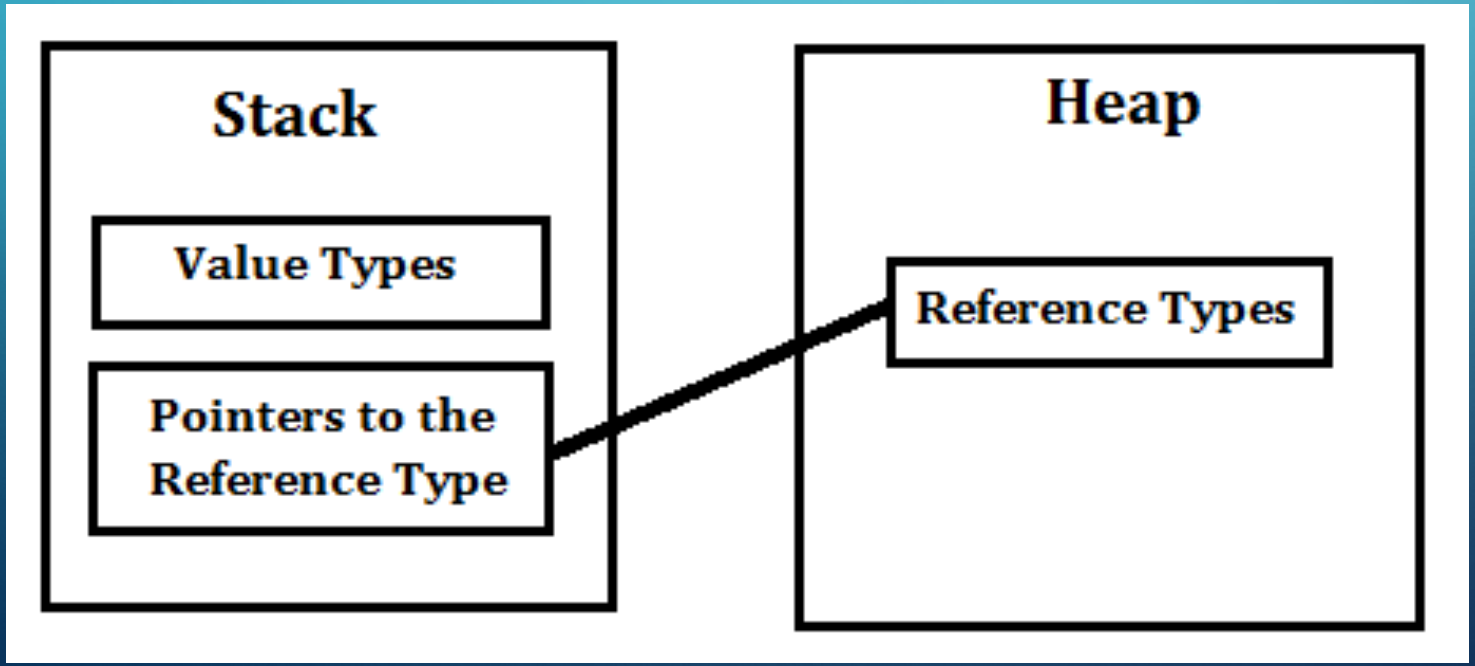
2. TYPES

There are two kinds of types in C#: *value types* and *reference types*.
***Variables of value types directly contain their data**
***variables of reference types store references to their data, being known as objects.**

These would be asked in the test.

value types and *reference types*

- **Reference Type** variables are stored in the heap
- **Value Type** variables are stored in the stack.

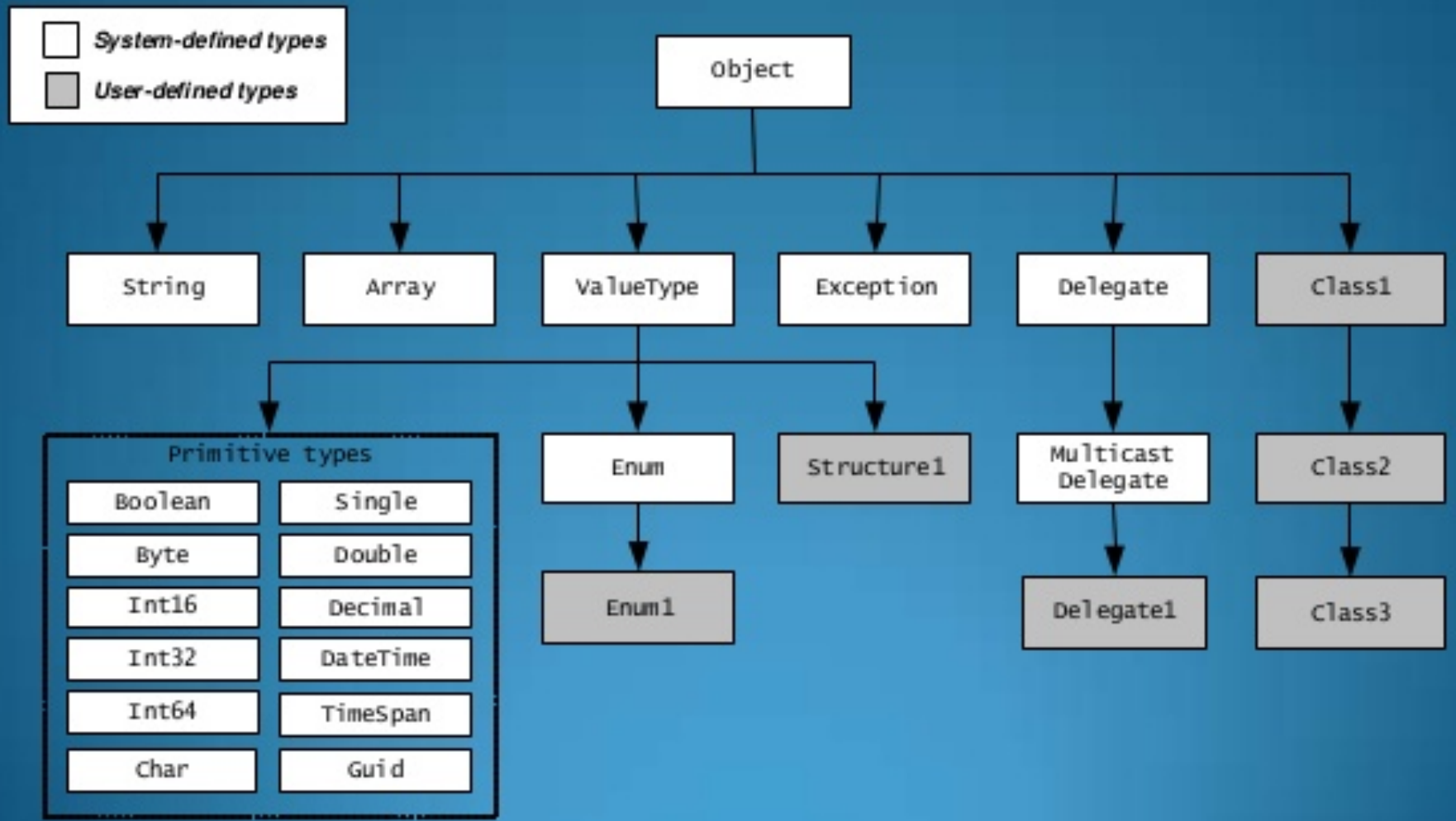


VALUE TYPES AND REFERENCE TYPES

With reference types:
it is possible for two variables to reference the same object
and thus possible for operations on one variable
to affect the object referenced by the other variable.

With value types:
the variables each have their own copy of the data,
and it is not possible for operations on one to affect the other
(except in the case of `ref` and `out` parameter variables).

Common Type System (CTS)



- All the intrinsic types are value types except for Object and String
- All user-defined types are reference types except for stucts

```
static void Main(string[] args)
{
    int i = 8;
    int j = i;
    j = 9;
    Console.WriteLine("int i = 8; int j = i; j = 9; What value is i now? " );
    Console.ReadLine();
    Console.WriteLine("i= " + i);
    Console.ReadLine();

    int[] A = { 1, 2, 3 };
    int[] B = A;
    B[0] = 4;
    Console.WriteLine("int[] A = { 1, 2, 3 };  int[] B = A;  B[0] = 4; What is A[0] now? ");
    Console.ReadLine();
    Console.WriteLine(" A[0] = " + A[0]);
    Console.WriteLine(" A[1] = " + A[1]);
    Console.WriteLine(" A[2] = " + A[2]);
    Console.ReadLine();

}
```

class System.String
Represents text as a series of Unicode characters.To browse the .NET Framework source code for this ty

Value Type

Reference Type

- These would be the test questions

VS LAB

- Test to change int and int[]

3. VALUE TYPES

- Simple Types
  - Signed integral: `sbyte`, `short`, `int`, `long`
  - Unsigned integral: `byte`, `ushort`, `uint`, `ulong`
  - Unicode characters: `char`
  - IEEE floating point: `float`, `double`
  - High-precision decimal: `decimal`
  - Boolean: `bool`
- Enum types
  - User-defined types of the form `enum E {...}`
- Struct types
  - User-defined types of the form `struct S {...}`
- Nullable value types

INITIALIZING VALUE TYPES

- Local variables in C# must be initialized before they are used.

```
int myInt;

myInt = 0;
```

VALUE TYPES LAB

- Simple types:
  - Console.WriteLine("int Max Value is " + int.MaxValue);
  - Console.WriteLine("int Min Value is " + int.MinValue);
- Enum:

  enum Day { Sat, Sun, Mon, Tue, Wed, Thu, Fri };

  Console.WriteLine("1st day is " + Day.Sat);

  Every enumeration type has an underlying type, which can be any integral type except char.

  Usually it is best to define an enum directly within a namespace so that all classes in the namespace can access it with equal convenience. However, an enum can also be nested within a class or struct.

VALUE TYPES LAB

- Struct
- typically used to encapsulate small groups of related variables, such as the coordinates of a rectangle

```
public struct Book
{
public decimal price;
public string title;
public string author;
}
```

NULLABLE VALUE TYPES

- Allow you to assign null to value **type** variables
- A nullable type can represent the correct range of values for its underlying value type, plus an additional `null` value.
- `Nullable<Int32>`, pronounced "Nullable of Int32,"
- int? For short (What is the default value? null)

4. REFERENCE TYPES

- Class types
  - Ultimate base class of all other types: `object`
  - Unicode strings: `string`
  - User-defined types of the form `class C {...}`
- Interface types
  - User-defined types of the form `interface I {...}`
- Array types
  - Single- and multi-dimensional, for example, `int[]` and `int[,]`
- Delegate types
  - User-defined types of the form `delegate int D(...)`

4. REFERENCE TYPES

- String

- the `string` type represents a sequence of UTF-16 code units.
- The + operator concatenates strings: string a = "good " + "morning";

The [] operator can be used for readonly access to individual characters of a `string`:

C#

```
string str = "test";
char x = str[2];  // x = 's';
```

String literals are of type `string` and can be written in two forms, quoted and @-quoted.

Quoted string literals are enclosed in double quotation marks ("):

C# Copy

```
"good morning"  // a string literal
```

String literals can contain any character literal. Escape sequences are included. The following example uses escape sequence `\\` for backslash, `\u0066` for the letter f, and `\n` for newline.

Copy

```
string a = "\\\u0066\n";
Console.WriteLine(a);
```

> **Note**
>
> The escape code `\udddd` (where `dddd` is a four-digit number) represents the Unicode character U+`dddd`. Eight-digit Unicode escape codes are also recognized: `\Udddddddd`.

Verbatim string literals start with @ and are also enclosed in double quotation marks. For example:

C# Copy

```
@"good morning"  // a string literal
```

The advantage of verbatim strings is that escape sequences are *not* processed, which makes it easy to write, for example, a fully qualified file name:

C#

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

To include a double quotation mark in an @-quoted string, double it:

C#

```
@""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

Strings are *immutable*--the contents of a string object cannot be changed after the object is created, although the syntax makes it appear as if you can do this. For example, when you write this code, the compiler actually creates a new string object to hold the new sequence of characters, and that new object is assigned to b. The string "h" is then eligible for garbage collection.

C# Copy

```
string b = "h";
b += "ello";
```

- each operation that appears to modify a String object actually creates a new string.
- Every time you use one of the methods in the System.String class, you create a new string object in memory, which requires a new allocation of space for that new object. In situations where you need to perform repeated modifications to a string, the overhead associated with creating a new String object can be costly.

SYSTEM.TEXT.STRINGBUILDER

- StringBuilder Class represents a mutable string of characters.
- Can be used when you want to modify a string without creating a new object. (In situations where you need to perform repeated modifications to a string)

5. VARIABLES

- A variable is nothing but a name given to a storage area that our programs can manipulate. (Variables represent storage locations)
- Each variable in C# has a specific type, which determines the size and layout of the variable's memory the range of values that can be stored within that memory and the set of operations that can be applied to the variable.
- There are several kinds of *variables*:
  - fields, array elements, local variables, and parameters.

DEFINING VARIABLES

- int i, j, k;
- char c, ch;
- float f;
- double d;

INITIALIZING VARIABLES

```
int d = 3, f = 5;    /* initializing d and f. */
int num;
num = Convert.ToInt32(Console.ReadLine());
```

6 TYPE CONVERSION

- **Implicit type conversion** — These conversions are performed by C# in a **type-safe** manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.
- **Explicit type conversion** — These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a **cast** operator.

IMPLICIT TYPE CONVERSION -

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

- If conversion is unavailable you'd know about it at compile time

EXPLICIT TYPE CONVERSION - CAST

```
double d = 5673.74;

int i;

// cast double to int.

i = (int)d;
```

BOXING AND UNBOXING – NEXT TIME

ASSIGNMENT 2 STEP2

- The codes for the console app have been changed a little bit because I found someone could not run it:

```
var city = Console.ReadKey().KeyChar;
```

- If you got errors and did research online, you may attach the links that helped you to get more cedits.