Data Types and Control Flow

C# Programming

Introduction to C#

- C# is Microsoft's premier language for .NET development.
- It is directly descended from two of the world's most successful computer languages: C and C++.
- C was invented by Dennis Ritchie in the 1970s.
- C++ was invented by Bjarne Stroustrup beginning in 1979
- Work on Java, which was originally called Oak, began in 1991 at Sun Microsystems.
- C# was created at Microsoft late in the 1990s, C#'s chief architect was Anders Hejlsberg.

Evolution of C#

- Since its original 1.0 release, C# has been evolving at a rapid pace.
- C# 2.0 was a watershed event in the lifecycle of C# because it added many new features, such as generics, partial types, and anonymous methods, that fundamentally expanded the scope, power, and range of the language.
- With the release of C# 3.0, Microsoft included lambda expressions, Language Integrated Query (LINQ), extension methods, and implicitly typed variables.
- With C# 4.0, two major feature first is the Task Parallel Library (TPL) and the second is Parallel LINQ (PLINQ) was introduced.

Version Summary

	C# 2.0	C# 3.0	C# 4.0	C# 5.0
Features added	• Generics • Partial types • Anonymous methods • Iterators • Nullable types • Private setters (properties) • Method group conversions (delegates) • Covariance and Contra-variance • Static classes	• Implicitly typed local variables • Object and collection initializers • Auto-Implemented properties • Anonymous types • Extension methods • Query expressions • Lambda expressions • Expression trees • Partial Methods	• Dynamic binding • Named and optional arguments • Generic co- and contravariance • Embedded interop types ("NoPIA")	• Asynchronous methods • Caller info attributes

Why C#?

- Power of C++ — C# brings with it object-oriented paradigm of C++. It implements the same through encapsulation, inheritance and polymorphism. It is as elegant as C++
- Productivity of Visual Basic — Promotes Rapid Application Development through simpler GUI programming with easy drag & drop support.
- Elegance of Java — No pointers (unless explicitly mentioned), fixed-size primary data types, Unicode characters, Type-safety, No multiple-inheritance, Garbage collection, Platform independence
- C# (pronounced as 'C Sharp') is a new object-oriented language which combines the power of C++ with productivity of Visual Basic and elegance of Java.

More on C#

- C# is the programming language that most directly reflects the underlying Common Language Infrastructure (CLI).
- C# code is made up of a series of statements, each of which is terminated with a semicolon.
- C# is a block-structured language, meaning statements are part of a block of code.
- One of the most important construct is a variable. It is a named memory location that can be assigned a value.

Presented by

Ranjan Bhatnagar

General Structure of a C# Program

- C# programs can consist of one or more files.
- Each file can contain zero or more namespaces.
- A namespace can contain types such as classes, structs, interfaces, enumerations, and delegates, in addition to other namespaces as:

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }
    struct YourStruct
    {
    }
    interface IYourInterface
    {
    }
 delegate int YourDelegate();
```

```
    enum YourEnum
    {
    }
    namespace YourNestedNamespace
    {
        struct YourStruct { }
    }
    class YourMainClass
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```

C# Coding Conventions

- The C# Language Specification does not define a coding standard.
- Coding conventions serve the following purposes:
  - They create a consistent look to the code, so that readers can focus on content, not layout.
  - They enable readers to understand the code more quickly by making assumptions based on previous experience.
  - They facilitate copying, changing, and maintaining the code.
  - They demonstrate C# best practices.

Layout Conventions

- Good layout uses formatting to emphasize the structure of your code and to make the code easier to read.
  - Use the default Code Editor settings
  - Write only one statement per line
  - Write only one declaration per line
  - If continuation lines are not indented automatically, indent them one tab stop
  - Add at least one blank line between method definitions and property definitions
  - Use parentheses to make clauses in an expression apparent

General Naming Conventions

- The general naming conventions discuss choosing the best names for the elements in your code.
  - Do choose easily readable identifier names
  - Do not use underscores, hyphens, or any other non-alphanumeric characters
  - Avoid using identifiers that conflict with keywords of widely used programming languages
  - Do not use abbreviations or contractions as parts of identifier names
  - Do use semantically interesting names rather than language-specific keywords for type names

Capitalization Conventions

- It is important to note that the common language runtime (CLR) supports case-sensitive and case-insensitive languages.
- Following Casing Styles are used:
  - Pascal : The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized.
  - Camel: The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized.
  - Uppercase : All letters in the identifier are capitalized.

General Rules for Identifiers

- When an identifier consists of multiple words, do not use separators, such as underscores ("_") or hyphens ("-"), between words. Instead, use casing to indicate the beginning of each word.
  - Do use Pascal casing for all public member, type, and namespace names consisting of multiple words.
  - Do use camel casing for parameter names.
  - Do not assume that all programming languages are case-sensitive.
  - Do not capitalize each word in so-called closed-form compound words.
  - Do capitalize both characters of two-character acronyms
  - Do capitalize only the first character of acronyms with three or more characters

Presented by

Ranjan Bhatnagar

Types

- C# is a strongly-typed language.
- Every variable and constant has a type.
- Even every expression that evaluates to a value has a type.
- Every method signature specifies a type for each input parameter and for the return value.
- A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.

Type can include

- The information stored in a type can include the following:
  - The storage space that a variable of the type requires.
  - The maximum and minimum values that it can represent.
  - The members (methods, fields, events, and so on) that it contains.
  - The base type it inherits from.
  - The location where the memory for variables will be allocated at run time.
  - The kinds of operations that are permitted.

Why Type?

- The compiler uses type information to make sure that all operations that are performed in your code are type safe.
- The compiler embeds the type information into the executable file as metadata.
- The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.
- When you declare a variable or constant in a program, you must either specify its type or use the var keyword to let the compiler infer the type.

Variable Declaration Syntax

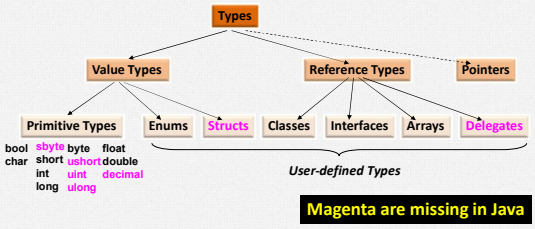
- Each variable needs to be declare.
- This means that you have to assign them a name and a type.
- C# syntax for declaring variables merely specifies the type and variable name:

  <type> <name>;
- There are an almost infinite number of types that you can use. This is because you can define your own types to hold whatever convoluted data you like.

C# Data types

- C# like C and C++ is a strongly typed language and so every variable must have a data type.
- Data types in C# are of 2 types
- Value Types
  - Built in types like int, char or user defined types created using struct or enum
  - For every value type there is a type in BCL
  - They are allocated in stack
- Reference Types
  - User defined type created using class or interfaces
  - Reference types are allocated in heap at runtime like pointers

C# Data Types



Magenta are missing in Java

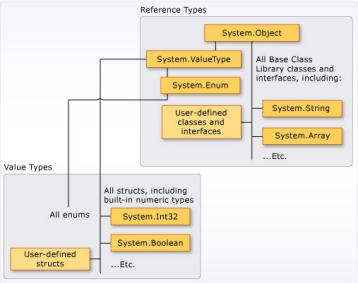
All types are compatible with *object*
- **can be assigned to variables of type *object***
- **all operations of type *object* are applicable to them**

Presented by

Ranjan Bhatnagar

Value & Reference Types in the CTS

- The following diagram shows the relationship between value types and reference types in the CTS



C# Value Types

- Boolean type
  - bool
- Integer types
  - byte int long sbyte short uint ulong ushort
- Character type
  - char
- Floating point types
  - decimal double float
- enum
- struct

More on Value Types

- Value types derive from System.ValueType, which derives from System.Object.
- Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared.
- There is no separate heap allocation or garbage collection overhead for value-type variables.
- There are two categories of value types: struct and enum.
- Value types are sealed means you cannot derive a type from any of the value types.

Boolean

- bool keyword is alias for System.Boolean.
- This datatype is CLS compliant (it can be used by any other .NET language)
- bool literal values are
  - true
  - false
- Example:

```
bool b = true;
Console.WriteLine(b); // prints True
b=false;
Console.WriteLine(b); // prints False
```

Integer Types

C# type	CLS compliant	System Type	Range
sbyte	No	System.Sbyte	-128 to 127 (signed 8-bit)
byte	Yes	System.Byte	0 to 255 (unsigned 8 bit)
short	Yes	System.Int16	-32768 to 32767 (signed 16 bit)
ushort	No	System.UInt16	0 to 65535 (unsigned 16 bit)
int	Yes	System.Int32	-2147783648(-2^{31}) to 2147483647(2^{31}-1) (signed 32 bit)
uint	No	System.UInt32	0 to 2^{32}-1(unsigned 32 bits)
long	Yes	System.Int64	-2^{63} to 2^{63}-1(signed 64 bit number)
ulong	No	System.UInt64	0 to 2^{64}-1(unsigned 64 bit number)

Integer Literal

- Integer literals can be written in two ways
  - Decimal: Example: 345, 2345678901
  - Hexadecimal where decimal numbers from 0 to 15 is represented as 0-9A-F: Example 0x11, 0XFF
- Integer literals types
  - Literal with no suffix (example 25) → could be int, uint, long, ulong
  - Literal with u or U as suffix (25u, 35U) → could be uint or ulong
  - Literal with l or L as suffix (6544425L, 76l) → could be long or ulong
  - Literal with ul or or uL or Ul or UL as suffix (25ul, 35UL) → is ulong

Floating point types

C# type	CLS compliant	System Type	Range
float	Yes	System.Single	$1.5*10^{-45}$ to $3.4*10^{38}$ (32 bit floating point number)
double	Yes	System.Double	$5.0*10^{-324}$ to $1.7*10^{308}$ (64 bit floating point number)
decimal	Yes	System.Decimal	±1.0 × 10−28 to ±7.9 × 1028 (28-29 significant digits)

Floating Point Literals

- Floating-point literals can be written in two ways
  - Fixed notation: 3.14
  - Scientific notation: 0.314E1, 314e-2
- Floating-point literal types
  - Literal with no suffix→ is double
  - Literal with F or f as suffix (Ex. 3.14f, 3.14F)→ is float
  - Literal with D or d as suffix (Ex. 3.14D, 3.14D) → is double
  - Literal with M or m as suffix (Ex. 3.14m, 3.14M) →is decimal
- decimal d = 3.45; //error
- decimal d = 3.45m; //ok
- float f = 3.4;// error
- float f = 3.4f;// ok

Unicode Character

- UNICODE is a 16 bit character.
- They are generally represented in hexadecimal format.
- "\u" in beginning of the character is used to represent hexadecimal character.
- The characters represented include all basic English letters, numbers, special characters and characters from other languages also !
- For example, Character 'A' represented in unicode as '\u0041' – which is the number 65 in base 10. Similarly Character '1' in \u0031'- which is 49.
- The Unicode Standard encodes characters in the range U+0000..U+10FFFF
- C# identifiers can be written in the form of unicode as well,
  - int \u0061; is same as int a;

Character type and Character Literal

- char keyword is alias for System.Char
- It is CLS compliant.
- The char keyword is used to declare a Unicode character. Range is u0000 to uffff (16 bit unicode character)
- Character literals can be written in the following forms
  - Any character within single quotes : Example '1','a'
  - Unicode escape sequence: Example '\u0041' represents the character 'A' in unicode
  - Hexadecimal escape sequence: Example '\x41' represents the character 'A' in unicode
  - Any escape sequence: Example \n for new-line character
- char c='a'; //ok
- char c=1; //error
- char c='\u0001'; //ok

Escape Characters

- \' -inserts a single quote into string literal
- \" -inserts a double quote
- \\ -inserts a backslash into string literal
- \a -triggers system alert(beep)
- \b -backspace
- \f -form feed
- \n -inserts a new line
- \r -inserts a carriage return
- \t -inserts a horizontal tab into the string literal
- \v -Vertical tab

Reference Type

- A type that is defined as a class, delegate, array, or interface is a reference type.
- The variable contains the value null until you explicitly create an instance of the object by using the new operator.
- When the object is created, the memory is allocated on the managed heap, and the variable holds only a reference to the location of the object.
- Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as garbage collection.

Presented by

Ranjan Bhatnagar

Strings

- A string is an object of type String whose value is text.
- The text is stored as a sequential read-only collection of Char objects.
- 'string' is alias for System.String.
- It is not a value type. It is a reference (a constant pointer to an object) type.
- Note that the equality operators (== and !=) are defined to compare the values of string objects, not references.

Facts about String

- There is no null-terminating character at the end of a C# string.
- C# string can contain any number of embedded null characters ('\0').
- The Length property of a string represents the number of Char objects it contains, not the number of Unicode characters.
- The String class provides many methods for safely creating, manipulating, and comparing strings.

More on Strings

- String literals can be written in two ways
- Regular string literal (enclosed within double quotes): Example: "C#", "C:\\My\\test" (uses escape char to escape the meaning of \)
- Verbatim string literals (@ preceding regular string literal). @ tells the compiler to interpret the character that appears between the double quote in exactly the same way as they are written. So instead of using escape char, a string can also be preceded by @.
- Example: @"C:\My\test", @" This is Madam's Pen"

Reading inputs from the console

- Console.ReadLine() Reads the next line of characters from the standard input stream.
- Console.Read() reads a character and returns in the form of int.

```
using System;
class SayHello
{
    public static void Main()
    {
        string s;
        Console.Write("Please your name");
        s = Console.ReadLine();
        Console.WriteLine("Hello " + s);
    }
}
```

Constants

- Constants are the variables whose values cannot change
- Keyword const is used.
- const int i=10, j=20;
- const string j = "hello";
- const float j = 1 / 2;
- int i = 1, j=2;
  const float k = i / j; //error
- const int i = 1, j=2;
  const float k = i / j;

Programming Constructs

Declaration	<data type> < variable name>;
	int MyInteger;
	string myString;
Assignment	myInteger = 17;
	myString = "\"myInteger\" is";
	MyFileStr = "C:\\Temp\\MyDir\\MyFile.doc"
	mystr = @"c:\temp\xyz.doc";
	myString = "It has a\nline break.";
Assignment Operator	= += -= *= /= %=
Output Variables	Console.WriteLine("{0} {1}.", myString, myInteger);
User Input	string userName;
	Console.WriteLine("Enter your name:");
	userName = Console.ReadLine();
	Console.WriteLine("Welcome {0}!", userName);
Operators	+ - * / % ++ --

Presented by

Ranjan Bhatnagar

Introducing Array Types

- Array types are reference type in C#.
- Array declaration and initialization

  int[ ] n1={20,10,5,13};
- Accessing array elements

  n1[0]
- Length of an array
- n1.Length

NOTE:

1. `int n1[]={20,10,5,13}; →error`
2. `int[5] n1;→error`
3. `int[] n1 ; n1[0]=4;→ error`

More on Arrays

- All arrays are reference types, even if their elements are value types.
- Arrays implicitly derive from the System.Array class
- System.Array class serve as a base class that provides methods for creating, manipulating, searching, and sorting arrays
- The length of an Array is the total number of elements it can contain.
- The rank of an Array is the number of dimensions in the Array.
- The lower bound of a dimension of an Array is the starting index of that dimension of the Array

Type Inference

- Visual C# 3.0, variables that are declared at method scope can have an implicit type var.
- These variables must be initialized. The type will be demined by the value that is assigned to them.
- That is why, these are called implicitly typed variables.
- var i=10; is same as int i=10;
- var name="Ram" is same as

  string name="Ram"

More on Type Inference

- Local variables can be given an inferred "type" of var instead of an explicit type.
- The var keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement.
- The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET Framework class library.
- The examples in the next slide show various ways in which local variables can be declared with var keyword.

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace CSharpProg
{
  class Program
  {
    static void Main(string[]
args)
    {
      // i is compiled as an int
      var i = 5;
      //s is compiled as string
      var s = "Hello";
      // a is compiled as int[]
      var a = new[] { 0, 1, 2 };
      string[] customers =
        { "Ram", "Shyam",
      "Ravi", "Hari", "Krishna",
      "Sameer" };
```

```
//expr is compiled as
IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr = from c in customers
  where c.StartsWith("R",
    StringComparison.
      CurrentCultureIgnoreCase)
    select c;
foreach (var item in expr)
   Console.WriteLine(item);
// anon is compiled as an
anonymous type
  var anon = new { Name = "Terry",
       Age = 34 };
// list is compiled as List<int>
          var list = new
List<int>();
        }
    }
}
```

Statements

- The C# code that comprises an application consists of statements made up of keywords, expressions and operators.
- Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition are statements.
- A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block.
- A statement block is enclosed in {} brackets and can contain nested blocks.

Presented by

Ranjan Bhatnagar

Expression

- You can create expressions by combining operators with variables and literal values.
- Both together referred to as operands when used with operators.
- Expressions can consist of a literal value, a method invocation, an operator and its operands, or a simple name.
- Simple names can be the name of a variable, type member, method parameter, namespace or type.

Few Example

```
using System;
namespace CSharpProg
{
    class Program
    {
        static void Main(string[] args)
        {
            // Expression statements.
            int i = 5;
            string s = "Hello World";
            // Invocation expression.
            DoWork();
            int num = 5;
            System.Console.WriteLine(num); // Output: 5
            num = 6;
            System.Console.WriteLine(num); // Output: 6
        }
        public static void DoWork()
        {
            //some code;
        }
    }
}
```

Advanced Expression

- Other than normal a few expressions could be kept in advanced category that we will consider in later presentations, such as:
  - Query Expression (LINQ)
    - Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language
  - Lambda Expression
    - A lambda expression is an anonymous function that can contain expressions and statements, and can be used to create delegates or expression tree types.
    - All lambda expressions use the lambda operator =>, which is read as "goes to".

Operators

- In C#, an operator is a program element that is applied to one or more operands in an expression or statement.
- Operators can be roughly classified into three categories:
  - Operators that take one operand, such as the increment operator (++) or new, are referred to as unary operators.
  - Operators that take two operands, such as arithmetic operators (+,-,*,/), are referred to as binary operators.
  - One operator, the conditional operator (?:), takes three operands and is the sole ternary operator in C#.

Arithmetic Operator

+ - / * ++ -- %

- Most of the operators work the same way as it does in C.
  - + , - are both unary and binary operators.
  - ++ and -- are unary pre and post increments.
  - / * and % are binary operators.
- Applicable to integer and floating point types

```
int i = 10;
int j = i+ 255;
byte k = i + 255; //error
const double  PI = 3.14;
double f = PI + 12;
```

Arithmetic operational errors

- When 2 of any integral types (other than long) are involved in arithmetic operation, the result is an int.
- byte i = 10;
- byte j = 20;
- j=j+i; //error
- But with compound assignment operators, this works because these operators perform the conversion.
- j+=i; //ok
- Similarly
- byte i = 10;
- i=i+1; //error
- But ++i; or i++; is ok
- Same is the case with other integral types expect int and long.

Presented by

Ranjan Bhatnagar

Operators -Relational

== != < > >= <=

- Relational operator return bool value – true or false.
  - int i=10;
  - int j=20;
- Console.WriteLine(i>j); // output is False
- Console.WriteLine(i == 10); // output is True

Operators - Bitwise

& | ^ ~

Operand 1	Operand 2	&	|	^	~ Operand 1
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Operators – Logical Short Circuit

&& || !

- These are both boolean operators. These work only on bool values.
- && and || are also called short circuit operators because they are optimized.
- && checks if the first condition is false. If it is so then it doesn't evaluate the second condition.
- || checks if the first condition is true. If it is so then it doesn't evaluate the second condition.
-

Shift Operators

>> <<

- If first operand is an int or uint (32-bit quantity), the shift count is given by the low-order five bits of second operand.
- If first operand is a long or ulong (64-bit quantity), the shift count is given by the low-order six bits of second operand.
- The high-order bits of first operand are discarded and the low-order empty bits are zero-filled. Shift operations never cause overflows.
- Console.WriteLine(2<<1); // prints 4
- Console.WriteLine(-1>>2); // prints -1
- Console.WriteLine(2 >>32); // prints 2
- Console.WriteLine(2 >> 33); // prints 1

Assignment Operators

= >= <= *= /= &= |= ^= <<= >>= %=

- int a = 10;
- int b=2;
- a+=b; // 12 same as a=a+b;
- Console.WriteLine(a);
- double d=45.3;
- d/=1.2;
- Console.WriteLine(d); // 37.75 same as d=d/1.2
- float d1=2.4f;
- d1 %= 1.2f;
- Console.WriteLine(d1); // 0

Other operators

Operator	Name	Example
?:	Ternary operator	10 % 2 > 0 ? 1 : 10
[]	Array subscript	a[1]
()	Cast	
??	Next Session	
::	namespace alias qualifier	Coming Up
.	dot operator	n.Length
is	Next Session	
as		
typeof	In reflection	
new	Next Session	
checked	Coming up	
unchecked		

Presented by

Ranjan Bhatnagar

Thanks