Core C# Language Features

Contents

1. Language essentials
2. Operators
3. Conditional statements
4. Loops

Demos folder:
Demos-02-CoreLanguageFeatures

1. Language Essentials

- Anatomy of a simple program
- Variables
- Constants
- Commonly used .NET data types
- Using the Console class
- Using the String class

Anatomy of a Simple Program

- The simplest type of program in .NET is a console application
  - Define a class
  - Implement a `static` `Main()` method
  - Optionally declare an array of `strings` as input to `Main()`
  - Optionally return an `int` from `Main()`

- Example

```
namespace HelloWorldApp
{
  class Program
  {
    static int Main(string[] args)
    {
      Console.WriteLine("Thanks for passing in {0} arguments!", args.Length);

      return 0;
    }
  }
}
```

Variables

- All programs use variables
  - Syntax: `type variableName = optional-initial-value;`
  - Example:
```
int yearsToRetirement = 20;
```
- Note:
  - Local variables (defined with a method) are uninitialized by default
  - You must initialize the variables before you use them

Constants

- Constants are fixed variables, cannot be changed
  - Syntax:

```
const type constantName = mandatory-compile-time-constant-value;
```

  - Example:

```
const double PI = 3.1415;
```

- Note:
  - There is also a `readonly` keyword
  - Useful for class members, allows you to perform run-time initialization within constructor, and constant thereafter

Commonly-Used .NET Data Types

C# keyword	System type	Description	Range
`bool`	`System.Boolean`	Truth or falsity	true, false
`sbyte`	`System.SByte`	Signed 8-bit integer	-128 to 127
`byte`	`System.Byte`	Unsigned 8-bit integer	0 to 255
`short`	`System.Int16`	Signed 16-bit integer	-32,768 to 32,767
`ushort`	`System.UInt16`	Unsigned 16-bit integer	0 to 65,535
`int`	`System.Int32`	Signed 32-bit integer	-2,147,483,648 to 2,147,483,647
`uint`	`System.UInt32`	Unsigned 32-bit integer	0 to +4,294,967,295
`long`	`System.Int64`	Signed 64-bit integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`ulong`	`System.UInt64`	Unsigned 64-bit integer	0 to 18,446,744,073,709,551,615
`char`	`System.Char`	Unicode character	U0000 to UFFFF
`float`	`System.Single`	32-bit floating-point number	±1.5E-45 to ±3.4E38
`double`	`System.Double`	64-bit floating-point number	±5.0E-324 to ±1.7E308
`decimal`	`System.Decimal`	96-bit signed number	±1.0E-28 to ±7.9E28
`string`	`System.String`	String of Unicode characters	Limited by system memory
`object`	`System.Object`	Base class for all .NET types	Can refer to any type of object

Using the Console Class (1 of 2)

- The `Console` class permits simple console I/O
  - `Console.Write()` – Outputs a message
  - `Console.WriteLine()` – Outputs a message and new-line
  - `Console.ReadLine()` – Reads a string from console
- Use `{0}` etc. as placeholders in output string
  - Parameters are matched to placeholders positionally
- Can use formatters to format output, e.g. `{0:c}`
  - See example on next slide

Using the Console Class (2 of 2)

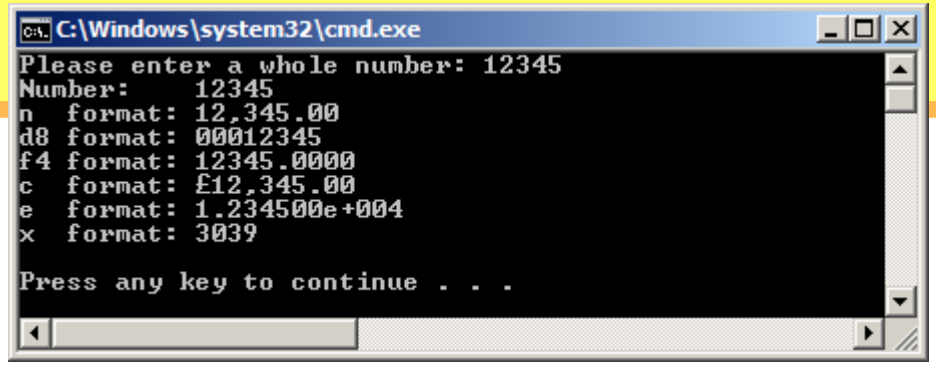
- Example of using `Console` (see `DemoConsole` project)

```
static void Main()
{
  // Prompt user to enter a numeric string.
  Console.Write("Please enter a whole number: ");
  string strNum = Console.ReadLine();

  // Parse numeric string, extract an int.
  int num = int.Parse(strNum);

  Console.WriteLine("Number:    {0}", num);       // Raw output
  Console.WriteLine("n  format: {0:n}", num);     // Number formatting (commas)
  Console.WriteLine("d8 format: {0:d8}", num);    // Decimal number formatting
  Console.WriteLine("f4 format: {0:f4}", num);    // Fixed-point formatting
  Console.WriteLine("c  format: {0:c}", num);     // Currency
  Console.WriteLine("e  format: {0:e}", num);     // Exponential
  Console.WriteLine("x  format: {0:x}", num);     // Hexadecimal

  Console.WriteLine();  // Blank line
}
```



Using the String Class (1 of 2)

- The `String` class represents a Unicode character string
  - String literals have the format `"xxx"`
  - Can contain escape characters (e.g. `\n`), see example on next slide
  - To prevent escape character expansion, prefix string literal with `@`
- Some useful properties/methods in the `String` class:
  - `Length`
  - `Compare()`, `Contains()`, `Equals()`
  - `Format()`, `Insert()`, `Remove()`, `Replace()`, `Split()`
  - `Trim()`, `PadLeft()`, `PadRight()`
  - `ToUpper()`, `ToLower()`
- String objects are immutable – you can't change them
  - If you have a lot of string manipulation, use `StringBuilder`

Using the String Class (2 of 2)

- Example of using `String` (see `DemoString` project)

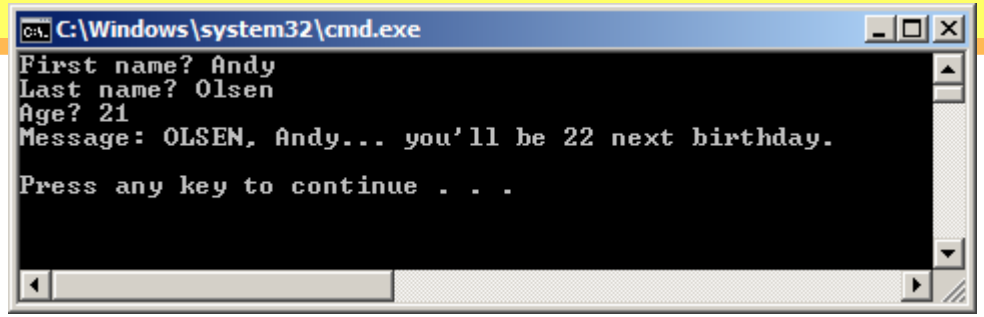
```
static void Main(string[] args)
{
  Console.Write("First name? ");
  String firstName = Console.ReadLine();

  Console.Write("Last name? ");
  String lastName = Console.ReadLine();

  Console.Write("Age? ");
  int age = int.Parse(Console.ReadLine());

  String message = string.Format("{0}, {1}... you'll be {2} next birthday.",
                                 lastName.ToUpper(), firstName, age + 1);

  Console.WriteLine("Message: {0}\n", message);
}
```



2. Operators

- Arithmetic operators
- Conditional operator
- Assignment operators
- Aside: working with strings
- Casting
- Relational operators
- Logical operators
- Bitwise operators

Arithmetic Operators

- Basic binary operators
  - `a + b` (addition)
  - `a - b` (subtraction)
  - `a * b` (multiplication)
  - `a / b` (division)
  - `a % b` (modulo, i.e. remainder)
- Basic unary operators
  - `+a` (unary plus)
  - `-a` (unary negation)
  - `a++` (postfix increment by 1)
  - `++a` (prefix increment by 1)
  - `a--` (postfix decrement by 1)
  - `--a` (prefix increment by 1)

Conditional Operator

- The conditional operator is like an in-situ if test
  - (*condition*) ? *trueResult* : *falseResult*

- Example:

```
bool isMale;
int age;
…
int togo = (isMale) ? (65 – age) : (60 – age);

Console.WriteLine("You have {0} years to go to retirement.", togo);
```

Assignment Operators (1 of 2)

- Basic assignment operator
  - a = b (assign b to a)
  - Performs widening conversion implicitly, if needed (see later)

- For value types (e.g. integers):
  - Assign LHS variable a copy of RHS value

```
int a = 100;
int b = 200;

b = 42;   // b is now 42.
a = b;    // a is now 42.
```

- For reference types (e.g. classes):
  - Assign LHS variable a reference to the RHS object

```
Person p1 = new Person("John", "Developer", 25000);
Person p2;

p2 = p1;   // p2 points to same Person object as p1.
```

Assignment Operators (2 of 2)

- Compound assignment operators:
  - a += b (calculate a + b, then assign to a)
  - a –= b (calculate a – b, then assign to a)
  - a *= b (calculate a * b, then assign to a)
  - a /= b (calculate a / b, then assign to a)
  - etc.

- Example:

```
// Use *= compound operator:
x *= a + b;

// Equivalent to the following (note the precedence):
x = x * (a + b);
```

Aside: Working with Strings

- String concatenation
  - `strResult = str1 + str2`
  - `strResult = str1 + obj`
- String shortcut concatenation
  - `str1 += str2`
  - `str1 += obj`
- Examples
  - What do the following statements do?

```
String message = "Hello";
int a = 5;
int b = 6;

Console.WriteLine(message + a + b);
Console.WriteLine(message + (a + b));
Console.WriteLine("" + a + b);
Console.WriteLine(a + b);
```

Casting

- Implicit conversions:
  - C# implicitly converts less-precise expns to more-precise expns
  - byte -> short -> int -> long -> float -> double
- Explicit conversions (aka casting):
  - You can explicitly cast an expression into a compatible other type
  - (*type*) *expression*
  - Might result in a loss of precision
- Explain the following example:

```
int judge1Score;
int judge2Score;
int judge3Score;
…
double averageScore = (double) (judge1Score + judge2Score + judge3Score) / 3;
```

- Questions:
  - What would happen without the above cast?
  - Can you achieve the same effect without using explicit casting?

Relational Operators (1 of 3)

- There are 6 relational operators (all return `boolean`):
  - `==` (equality)
  - `!=` (inequality)
  - `>` (greater-than)
  - `>=` (greater-than-or-equal)
  - `<` (less-than)
  - `<=` (less-than-or-equal)

- Type checking
  - `is` (tests if variable is instance of given class/interface)

Relational Operators (2 of 3)

- If you use == or != on value types (e.g. integers):
  - You are comparing numeric values
  - i.e. do they contain the same value
- If you use == or != on reference-types objects:
  - You're comparing object references: do they point to same object?
  - Note: `String` supports value-based == and != operators
- To compare the *values* of reference-type objects:
  - Use the `Equals()` method, e.g. `account1.Equals(account2)`

Logical Operators

- Short-circuit logical operators:
  - && (logical AND)
  - || (logical OR)
- Logical inverse operator:
  - ! (logical NOT)
- Note:
  - All these operators require bool operands

Bitwise Operators

- Bitwise AND and OR binary operators
  - & (bitwise AND)
  - ^ (bitwise exclusive OR)
  - | (bitwise inclusive OR)
- Bitwise NOT unary operator
  - ~ (bitwise NOT)
- Bitwise shift operators
  - << (shift bits left)
  - >> (shift bits right)

3. Conditional Statements

- Using if tests
- Quiz
- Nesting if tests
- Using switch tests

Using if Tests

- Basic if tests

```
if (booleanTest) {
  body
}
```

Executes *body* if *booleanTest* is `true`

- if-else tests

```
if (booleanTest) {
  body1
} else {
  body2
}
```

Executes *body1* if *booleanTest* is `true`

Otherwise, executes *body2*

- if-else-if tests

```
if (booleanTest1) {
  body1
}
else if (booleanTest2) {
  body2
}
else if (test3) {
  body3
}
…
else {
  lastBody
}
```

Executes *body1* if *booleanTest1* is `true`

Or executes *body2* if *booleanTest2* is `true`

Or executes *body3* if *booleanTest3* is `true`

If all else fails, executes (optional) *lastBody*

- Notes:
  - Test conditions must be `bool`
  - {} are optional if you want a 1-line statement

Quiz

- Explain the following examples
  - … and spot the deliberate gotchas ☺

```
int i = … ;
int j = … ;

if (i == j) {
 Console.WriteLine("Equal.");
}

if (i == j)
  Console.WriteLine("Equal.");

if (i == j)
  Console.WriteLine("Equal.");
  Console.WriteLine("Goodbye.");

if (i == j);
  Console.WriteLine("Equal.");

if (i = j)
  Console.WriteLine("Equal.");

if (i)
  Console.WriteLine("i is non-zero.");
```

```
bool b = … ;

if (b == true) …

if (b == methodThatReturnsBool()) …

if (b = methodThatReturnsBool()) …
```

```
bool b1 = … ;
bool b2 = … ;

if (b1)
if (b2)
Console.WriteLine("Yes");
else Console.WriteLine("No");
```

Nesting if Tests

- You can nest if tests inside each other
  - Use {} to ensure correct logic, as needed
  - Use indentation for readability

```
int age = … ;
String gender = … ;

if (age < 18) {

  if (gender == "Male") {
    Console.WriteLine("boy");
  } else {
    Console.WriteLine("girl");
  }

} else {

   if (age >= 100) {
     Console.WriteLine("centurion ");
   }

  if (gender == "Male") {
    Console.WriteLine("man");
  } else {
    Console.WriteLine("woman");
  }
}
```

Using switch Tests

- The `switch` statement is useful if you want to test a single expression against a finite set of expected values

- General syntax:

```
switch (expression) {

case constant1:
  branch1Statements;
  break;

case constant2:
  branch2Statements;
  break;

…

default:
  defaultBranchStatements;
  break;
}
```

- Expression can be:
  - integral values
  - characters or strings
  - enums
- Cases:
  - Must be (different) constants
- If you omit `break`:
  - Compilation error occurs
- The `default` branch:
  - Is optional
  - Doesn't have to be at the end!

4. Loops

- Using while loops
- Using do-while loops
- Using for loops
- Using foreach loops
- Unconditional jumps

Using while Loops

- The `while` loop is the most straightforward loop construct
  - Boolean test is evaluated
  - If `true`, loop body is executed
  - Boolean test is re-evaluated
  - Etc…

```
while (booleanTest) {
   loopBody
}
```

- Note:
  - Loop body will not be executed if test is `false` initially

- How would you write a `while` loop…
  - To display 1 – 5?
  - To display the first 5 odd numbers?
  - To read 5 strings from the console, and output in uppercase?

Using do-while Loops

- The `do-while` loop has its test at the end of the loop
  - Loop body is always evaluated at least once
  - Handy for input validation
  - Note the trailing semicolon!

```
do {
   loopBody
} while (booleanTest);
```

- How would you write a `do-while` loop…
  - To keep reading strings from the console, until the user enters "Oslo", "Bergen", or "Trondheim" (in any case)?

Using for Loops

- The `for` loop is the most explicit loop construct
  - Initialization part can declare/initialize variable(s)
  - Test part can incorporate any number of tests
  - Update part can do anything, e.g. update loop variable(s)
  - You can omit any (or all!) parts of the for-loop syntax

```
for (init; booleanTest; update) {
  loopBody
}
```

Note:
If you declare variables in the initialization section (or in the loop body, of course), they are scoped to the for-loop

- How would you write a `for` loop…
  - To display the first 5 odd numbers?
  - To display 100 – 50, in downward steps of 10?
  - To loop indefinitely?

Using foreach Loops

- The `foreach` loop iterates through an array or collection
  - Often simpler than a conventional `for` loop

- Examples
  - Iterate through an array of integers

```
int[] examMarks = { 75, 85, 92, 71, 99 };

foreach (int m in examMarks)
{
  Console.WriteLine("Mark: {0}", m);
}
```

  - Iterate through a collection of strings

```
List<string> favouriteCities = new List<string>();
…

foreach (string c in favouriteCities)
{
  Console.WriteLine("City: {0}", c);
}
```

Unconditional Jumps (1 of 2)

- Sometimes it can be convenient to use unconditional jump statements within a loop
  - `break`
    - Terminates innermost loop
  - `continue`
    - Terminates current iteration of innermost loop, and starts next iteration
    - If used in a `for` loop, transfers control to the update part
  - `return`
    - Terminates entire method
- Discuss:

```
for (initialization; test; update) {
  …
  if (someCondition)
    break;
  …
  if (someOtherCondition)
    continue;
  …
}
```

Unconditional Jumps (2 of 2)

- You can use `break` and `continue` in nested loops
  - By default, they relate to the inner loop
  - To relate to the outer loop, use labels
- Discuss:

```
myOuterLabel:

// Outer Loop.
for (init; booleanTest; update) {
  …

  // Inner Loop.
  for (init; booleanTest; update) {
    if (someCondition)
      break myOuterLabel;
    …
    if (someOtherCondition)
      continue myOuterLabel;
    …
  }
}
```

Any Questions?