C#
Fundamentals

Identifiers

- An identifier is a name used to identify a class, variable, method, or any other user-defined item.
- The basic rules for naming identifiers in C# are as follows:
  - A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore.
  - The first character in an identifier cannot be a digit.
  - It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \.
    - However, an underscore (_) can be used.
  - It should not be a C# keyword.

Keywords

- Keywords are reserved words predefined to the C# compiler.
- These keywords cannot be used as identifiers.
  - However, if you want to use these keywords as identifiers, you may prefix the keyword with the `@` character.

- In C#, some identifiers have special meaning in context of code, such as `get` and `set`, these are called contextual keywords.

Keywords

Reserved Keywords						
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	in	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	while					

Keywords

Contextual Keywords						
add	alias	ascending	descending	dynamic	from	get
global	group	into	join	let	orderby	partial (type)
Partial (method)	remove	select	set			

Naming Conventions

- Variable names follow camel case notation.
- Start the names of variables with a lowercase letter and capitalize the first letter of each word after the first word.
- **Examples:**

```
hourlyRate, num, amountAfterDiscount
```

- All other identifiers follow Pascal case notation.
  - Class, method, property, Constant etc.
- Capitalize the first letter of each word.
- **Examples:**

```
Employee, GetArea, DaysInJanuary
```

- **C# Coding Standards and Naming Conventions:**
  - https://github.com/ktaranov/naming-convention/blob/master/C%23%20Coding%20Standards%20and%20Naming%20Conventions.md

Initialization of Variables

- The C# compiler requires that any variable be initialized with some starting value before you refer to that variable in an operation.

- Variables that are fields in a class, if not initialized explicitly, are by default zeroed out when they are created.
  - `null` for reference types, zero for numeric data types, and `false` for bools.
- Variables that are local to a method must be explicitly initialized in your code prior to any statements in which their values are used.

Literal Values

- By default, a literal value having a decimal point is considered as `double` by C# compiler.
- To identify literal values as `float` values, you must type the letter `f` or `F` after the number.
- To identify `decimal` values, you must type the letter `m` or `M` after the number.
- Uppercase `F` or `M` is recommended.

- **Note:** If you omit the letter, you'll get build error when you try to run the project.

Declare and Initialize Variables

- Syntax:

```
datatype variableName = value;
```

- Examples:

```
int myCounter = 1;
long numberOfBytes = 20000L;      // L or l indicates a long value
float interestRate = 8.125F;      // F or f indicates a float value
double price = 14.95;
decimal total = 24218.192M;       // M or m indicates a decimal value
char letter = 'A';                // enclose a character value in single quotes
bool isValid = false;
int x = 0, y = 0;                 // initialize 2 variables in 1 statement
```

Declare and Initialize a Constant

- Syntax:

```
const datatype ConstantName = value;
```

- Examples:

```
const int DaysInNovember = 30;
```

Declare and Initialize a String

- Example:

```
string msg1 = "Invalid data entry";
string msg2 = "";
string msg3 = null;
```

Concatenate Strings

- Example:

```
string firstName = "John";                      // firstName is "John"
string lastName = "Smith";                      // lastName is "Smith"
string fullName = firstName + " " + lastName; // fullName is John Smith
```

- + operator is used to concatenate strings.

Concatenate a String and a Number

- Example:

```
double price = 14.95;
string priceString = "Price: $" + price;     // priceString is "Price: $14.95"
```

- When a numeric datatype is concatenated with a string, it is implicitly converted to string and then concatenation takes place.

Type Inference

- Type inference makes use of the `var` keyword.
- The compiler "infers" what the type of the variable is by what the variable is initialized to.
- For example:

```
var someNumber = 10;
```

- Even though `someNumber` is never declared as being an `int`, the compiler figures this out and `someNumber` remains as an `int` for as long as it is in scope.

Type Inference - Example

- Here is a short program to demonstrate:

```
var name = "Bugs Bunny";
var age = 25;
var isRabbit = true;

Console.WriteLine("name is type " + name.GetType());
Console.WriteLine("age is type " + age.GetType());
Console.WriteLine("isRabbit is type " + isRabbit.GetType());
```

Output:

```
name is type System.String
age is type System.Int32
isRabbit is type System.Bool
```

Type Inference

- There are few rules that you need to follow:
  - The variable must be initialized. Otherwise, the compiler doesn't have anything from which to infer the type.
  - Cannot be initialized to null.

- After the variable has been declared and the type inferred, the variable's type cannot be changed.
- When established, the variable's type follows all the strong typing rules that any other variable type must follow.

Common Escape Sequences

Key	Description
\n	New line
\t	Horizontal Tab
\\	Backslash
\"	Quotation

- For complete list:
  https://docs.microsoft.com/en-us/cpp/c-language/escape-sequences?view=vs-2019

Common Escape Sequences - Examples

- Example:

```
string code = "JSPS";
double price = 49.50;
string result = "Code: " + code + "\n" + "Price: $" + price; // Code: JSPS
                                                                Price: $49.50

string path = "c:\\c#.net\\files";      // c:\c#.net\files
string msg = "Type \"x\" to exit";      // Type "x" to exit
```

Verbatim String Literals

- To code a verbatim string literal, you can place an @ sign before the opening double quote of a string.
- Within the string, you can enter backslashes, and other special characters without using the escape sequences.
- Basically the @ symbol tells the string constructor to ignore escape characters and line breaks.
- However, to enter a double quote within a verbatim string literal, you must enter two double quotes.

```
string str = @"hello \n world";          // hello \n world
string path = @"c:\c#.net\files";        // c:\c#.net\files
string msg = @"Type ""x"" to exit";      // Type "x" to exit
```

String Interpolation

- This feature inserts values of variables into a string with simple syntax.

```
string name = "John";
int age = 35;

// string concatenation
string s1 = name + " is " + age + " years old";

// string interpolation
string s2 = $"{name} is {age} years old";
```

String Interpolation

- You can set number of places required for the value to be displayed on the screen.
- A positive number right aligns the text, leaving blank spaces on the left.
- Negative number left aligns the text, leaving blank spaces on the right.

```
string firstname = "John";
string lastname = "Smith";
string fullname = $"{firstname, 10} {lastname, -10}";
```

- Output:

Value Types and Reference Types

- C# distinguishes between two categories of data type:
  - Value types
  - Reference types

- Conceptually, the difference is that a value type stores its value directly, whereas a reference type stores a reference to the value.
- It is important to be aware of whether a type is a value type or a reference type because of the different effect each assignment has.

Value Types and Reference Types

- For example, `int` is a value type, which means that the following statement results in two locations in memory storing the value `20`:

```
int i = 20;
int j = i;
```

- However, consider the following example. For this code, assume you have defined a class called `Circle`; and that `Circle` is a reference type and has an `int` member variable called `radius`:

```
Circle c1, c2;
c1 = new Circle();
c1.radius = 30;      // radius is a field defined in Circle class
c2 = c1;

Console.WriteLine(c2.radius);
c2.radius = 50;
Console.WriteLine(c1.radius);
```

Value Types and Reference Types

- The crucial point to understand is that after executing this code, there is only one `Circle` object.
- `c1` and `c2` both point to the memory location that contains this object.
- Because `c1` and `c2` are variables of a reference type, declaring each variable simply reserves a reference.
  - It doesn't instantiate an object of the given type.
- In neither case is an object actually created.
- To create an object, you have to use the `new` keyword.
- Because `c1` and `c2` refer to the same object, changes made to `c1` will affect `c2` and vice versa.
- Hence, the code will display `30` and then `50`.

```
Circle c1, c2;
c1 = new Circle();
c1.radius = 30;
c2 = c1;

Console.WriteLine(c2.radius);
c2.radius = 50;
Console.WriteLine(c1.radius);
```

Predefined Value Types

Type	Represents	Range	Default Value
`bool`	Boolean value	True or False	False
`byte`	8-bit unsigned integer	0 to 255	0
`char`	16-bit Unicode character	U +0000 to U +ffff	'\0'
`decimal`	128-bit precise decimal values with 28-29 significant digits	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / 10^{0 \text{ to } 28}$	0.0M
`double`	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324}$ to $(+/-)1.7 \times 10^{308}$	0.0D
`float`	32-bit single-precision floating point type	-3.4×10^{38} to $+ 3.4 \times 10^{38}$	0.0F
`int`	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
`long`	64-bit signed integer type	-923,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
`sbyte`	8-bit signed integer type	-128 to 127	0
`short`	16-bit signed integer type	-32,768 to 32,767	0
`uint`	32-bit unsigned integer type	0 to 4,294,967,295	0
`ulong`	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
`ushort`	16-bit unsigned integer type	0 to 65,535	0

Predefined Reference Types

- C# supports two predefined reference types, `object` and `string`:

- `object` The root type.
  All other types (including value types) are derived from `object`.

- `string` Unicode character `string`.

The `object` Type

- In C#, the `object` type is the ultimate parent type from which all other basic and user-defined types are derived.
- The `object` type implements a number of basic, general-purpose methods, such as `Equals()` and `ToString()`.
- User-defined classes may need to provide replacement implementations of some of these methods using an object-oriented technique known as overriding.
- When you override `ToString()`, for example, you equip your class with a method for intelligently providing a string representation of itself.

The `string` Type

- C# recognizes the `string` keyword, which under the hood is translated to the .NET class, `System.String`.
- With it, operations like string concatenation and string copying are a snap:

```
string str1 = "Hello ";
string str2 = "World";
string str3 = str1 + str2; // string concatenation
```

- Despite this style of assignment, `string` is a reference type.

The `string` Type

- Behind the scenes, a `string` object is allocated; and when you assign one string variable to another string, you get two references to the same string in memory.
- However, string differs from the usual behavior for reference types.
- For example, strings are immutable.
- Making changes to one of these strings creates an entirely new string object, leaving the other string unchanged.

The `string` Type

- Consider the following code:

```
string s1 = "a string";
string s2 = s1;

Console.WriteLine("s1 is " + s1);
Console.WriteLine("s2 is " + s2);

s1 = "another string";

Console.WriteLine("s1 is now " + s1);
Console.WriteLine("s2 is now " + s2);
```

Output:

```
s1 is a string
s2 is a string
s1 is now another string
s2 is now a string
```

The `string` Type

- Changing the value of `s1` has no effect on `s2`, contrary to what you'd expect with a reference type!
- What's happening here is that when `s1` is initialized with the value `a string`, a new string object is allocated.
- When `s2` is initialized, the reference points to this same object, so `s2` also has the value `a string`.
- However, when you now change the value of `s1`, instead of replacing the original value, a new object is allocated for the new value.
- The `s2` variable will still point to the original object, so its value is unchanged.
- In general, the `string` class has been implemented so that its semantics follow what you would normally intuitively expect for a string.

C# - Type Conversion

- Type conversion is basically type casting or converting one type of data to another type.
- In C#, type casting has two forms:
  - **Implicit Type Conversion** - these conversions are performed by C# in a type-safe manner. Examples are conversions from smaller to larger integral types.
  - **Explicit Type Conversion** - these conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

C# - Type Conversion

- Implicit Type Conversion Example:

```
int i = 10;
double d = i;                    // int implicitly converts to double
string s = "i = " + i;           // int implicitly converts to string
```

- Explicit Type Conversion Example:

```
double d = 10.5;
int i = (int)d;          // double is explicitly converted to int
```

Class Convert - Type Conversion Methods

#	Methods & Description	
1	`ToBoolean`	- Converts a type to a Boolean value, where possible.
2	`ToByte`	- Converts a type to a byte.
3	`ToChar`	- Converts a type to a single Unicode character, where possible.
4	`ToDateTime`	- Converts a type (integer or string type) to date-time structures.
5	`ToDecimal`	- Converts a floating point or integer type to a decimal type.
6	`ToDouble`	- Converts a type to a double type.
7	`ToInt16`	- Converts a type to a 16-bit integer.
8	`ToInt32`	- Converts a type to a 32-bit integer.
9	`ToInt64`	- Converts a type to a 64-bit integer.
10	`ToSbyte`	- Converts a type to a signed byte type.
11	`ToSingle`	- Converts a type to a small floating point number.
12	`ToString`	- Converts a type to a string.
13	`ToType`	- Converts a type to a specified type.
14	`ToUInt16`	- Converts a type to an unsigned int type.
15	`ToUInt32`	- Converts a type to an unsigned long type.
16	`ToUInt64`	- Converts a type to an unsigned big integer.

Class Convert - Type Conversion Methods

- Example:

```
string s = "10";

int i = Convert.ToInt32(s);      // convert string to int
s = i.ToString();                // convert int to string
```

Standard Numeric Formatting Codes

Code	Format	Description
`C or c`	Currency	Formats the number as currency with the specified number of decimal places.
`P or p`	Percent	Formats the number as a percent with the specified number of decimal places.
`N or n`	Number	Formats the number with thousands separators and the specified number of decimal places.
`F or f`	Float	Formats the number as a decimal with the specified number of decimal places.
`D or d`	Digits	Formats an integer with the specified number of digits.
`E or e`	Exponential	Formats the number in exponential notation with the specified number of decimal places.
`G or g`	General	Formats the number as a decimal or in scientific notation depending on which is more compact.

ToString() Method to Format a Number

- Example:

```
decimal amount = 1547.2M;
Console.WriteLine(amount.ToString("C"));          // $1,547.20

decimal interest = 0.023M;
Console.WriteLine(interest.ToString("P1"));       // 2.3 %

int quantity = 15000;
Console.WriteLine(quantity.ToString("N0"));       // 15,000

double payment = 432.8175;
Console.WriteLine(payment.ToString("F3"));        // 432.818
```

Nullable Data Types

- A nullable data type is a value data type that can store a `null` value.
- By default, value data types (`int`, `decimal`, `bool` etc.) can't store `null` values.
- In contrast, reference data types (`string`, `class`) can store `null` values.
- To declare a nullable data type, code a `?` immediately after the keyword for the value data type.

- **Note:** If you use a variable with a nullable data type in an arithmetic expression and the value of the variable is `null`, the result of the arithmetic expression is always `null`.

Nullable Data Types - Example

- Example:

```
int? quantity;
quantity = null;
quantity = 0;
quantity = 20;

decimal? salesTotal = null;

bool? isValid = null;

string? msg = null;   // not allowed
```

C# - Operators

- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- C# is rich in built-in operators and provides the following type of operators:
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Bitwise Operators
  - Assignment Operators
  - Misc. Operators

Arithmetic Operators

Operator	Description	Example (A=10, B=20)
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator increases integer value by one	A++ will give 11
--	Decrement operator decreases integer value by one	A-- will give 9

Relational Operators

Operator	Description	Example (A=10, B=20)
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

Operator	Description	Example (A=true, B=false)
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.
^	Called Logical XOR Operator. If both the operands are equal, then condition becomes false. If one of the operand is true, then the condition is true.	(A ^ B) is true.

Bitwise Operators

Operator	Description	Example (Assume A = 60 and B = 13)
&	Binary AND Operator copies a bit to the result if it exists in both operands	(A & B) = 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand	(A | B) = 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits	(~A) = -61, which is 1100 0011 in 2's complement due to a signed binary number
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand	A << 2 = 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand	A >> 2 = 15, which is 0000 1111

Assignment Operators

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

Misc. Operators

Operator	Description	Example
`sizeof()`	Returns the size of a data type.	sizeof(int), will return 4.
`typeof()`	Returns the type of a class.	typeof(int); returns System.Int32
`? :`	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y
`is`	Determines whether an object is of a certain type.	If (ford is Car) // checks if ford is an object of the Car class.
`as`	Cast without raising an exception if the cast fails.	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

Exercise

- Write an app that applies the discount percent to subtotal to find the total payable amount.
- Take the subtotal and discount percent from the user.
- Format the output to currency.
- Here is a sample run:

```
Output:

Enter subtotal: 220
Enter discount percent (0 - 100): 20

Discounted Amount: $44.00
Total: $176.00
```

Exercise - Solution

```
Console.Write("Enter subtotal: ");
double subtotal = double.Parse(Console.ReadLine());

Console.Write("Enter discount percent (0 - 100): ");
double discountPercent = double.Parse(Console.ReadLine());

double discountedAmount = subtotal * discountPercent / 100;
double total = subtotal - discountedAmount;

Console.WriteLine("\nDiscounted Amount: " + discountedAmount.ToString("C"));
Console.WriteLine("Total: " + total.ToString("C"));
```

Output:

```
Enter subtotal: 220
Enter discount percent (0 - 100): 20

Discounted Amount: $44.00
Total: $176.00
```

if Statement

- Example:

```
if (subTotal >= 100)
{
      discountPercent = 0.2;
}
```

if-else Statement

- Example:

```
if (subTotal >= 100)
{
      discountPercent = 0.2;
}
else
{
      discountPercent = 0.1;
}
```

if-else-if Statements

- Example:

```
if (subTotal >= 100 && subTotal < 200)
      discountPercent = 0.2;
else if (subTotal >= 200 && subTotal < 300)
      discountPercent = 0.3;
else if (subtotal >= 300)
      discountPercent = 0.4;
else
      discountPercent = 0.1;
```

switch Statements

- Example:

```
switch (expression)
{
      case 1:
            Console.WriteLine("Case 1");
            break;
      case 2:
            Console.WriteLine("Case 2");
            break;
      default:
            Console.WriteLine("Default case");
            break;
}
```

Exercise

- Create an app that calculates the total amount based on the type of customer and subtotal amount.
- If the Customer type is R:
  - If the Subtotal is greater than or equal to 250, the Discount percent is 25%.
  - If the Subtotal is greater than or equal to 100 and less than 250, the Discount percent is 10%.
  - If the Subtotal is less than 100, the Discount percent is 0%.
- If the Customer type is C:
  - If the Subtotal is greater than or equal to 250, the Discount percent is 30%.
  - If the Subtotal is less than 250, the Discount percent is 20%.
- The Discount Percent should display the percent (%) sign.
- The Discounted Amount and Total should display the dollar ($) sign.
- Here is a sample run:

Output:

```
Enter customer type: R
Enter subtotal: 150

Discount percent: 10 %
Discounted Amount: $15.00
Total: $135.00
```

while Loop

- Example:

```
int i = 1, sum = 0;

while (i < 5)
{
      sum += i;
      i++;
}
```

do-while Loop

- Example:

```
int x = 0;

do
{
      Console.WriteLine(x);
      x++;
} while (x < 5);
```

for Loop

- Example:

```
for (int i = 1; i <= 5; i++)
{
      Console.WriteLine(i);
}

// for loop that adds the numbers 8, 6, 4, and 2
int sum = 0;
for (int j = 8; j > 0; j-=2)
{
      sum += j;
}
```

Exercise

- Create an app that calculates the future value of the investment.
- To calculate a future value, the user must enter the monthly investment amount, the yearly interest rate, and the number of years the investment will be made.

- **Hint:**
  - Convert Yearly Interest Rate to Monthly Interest Rate.
  - Convert Number of Years to Number of Months.
  - Formula to calculate Future Value:

```
for (int i=0; i<numberOfMonths; i++)
{
    futureValue = (futureValue + monthlyInvestment) * (1 + monthlyInterestRate);
}
```

Output:

```
Enter monthly investment: 100
Enter yearly interest rate: 7.5
Enter number of years: 10

Future value: $17,904.24
```

Do It Yourself!

- **Sales Commission Calculator:** A large company pays its salespeople on a commission basis.
- The salespeople receive $200 per week plus 9% of their gross sales for that week.
- For example, a salesperson who sells $5,000 worth of merchandise in a week receives $200 plus 9% of $5,000, or a total of $650.
- You've been supplied with a list of the items sold by each salesperson.
- The values of these items are as follows:

Item	Value
1	239.99
2	129.75
3	99.95
4	350.89

- Develop a C# app that inputs one salesperson's items sold for the last week, then calculates and displays that salesperson's earnings.
- There's no limit to the number of items that can be sold by a salesperson.

 Do It Yourself!

- **Salary Calculator:** Develop a C# app that will determine the gross pay for three employees.
- The company pays straight time for the first 40 hours worked by each employee and time-and-a-half for all hours worked in excess of 40 hours.
- You're given a list of the three employees of the company, the number of hours each employee worked last week and the hourly rate of each employee.
- Your app should input this information for each employee, then should determine and display the employee's gross pay.

Thank You

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.

References

Material has been taken from:

- Visual C# 2012: How to Program:
- https://learning.oreilly.com/library/view/visual-c-2012/9780133380170/ch03.html

- Professional C# 7 and .NET Core 2.0:
- https://learning.oreilly.com/library/view/professional-c-7/9781119449270/c02.xhtml