Module 01

Data Types

All applications store and manipulate data within the computer's memory. C# supports two kinds of data types used to represent real-world information. Value types are so-called because they contain the actual value of the data they store. For example, you might have an int type that stores the value 3. The literal value of 3 is stored in the variable that you declare to hold it.

With the exception of DateTime and string, in the below table, the data types listed are aliases for structs in .NET that represent the data types in the Microsoft .NET Framework. Anyplace you can use int you can also use System.Int32. We'll cover structs in module four.

Reference types are also known as objects. Reference types are created from class files, which you will cover in module five A reference type stores a reference to the location in memory of the object. . If you are familiar with C/C++ then you can think of a reference to the memory location to be the same as a pointer. C# does not require you to use pointers.

The following table shows the most commonly used value types.

Type	Description	Size (bytes)	.NET Type	Range
int	Whole numbers	4	System.Int32	-2,147,483,648 to 2,147,483,647
long	Whole numbers (bigger range)	8	System.Int64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	Floating-point numbers	4	System.Single	+/-3.4 x 10^38
double	Double precision (more accurate) floating-point numbers	8	System.Double	+/-1.7 x 10^308

decimal	Monetary values	16	System.Decimal	28 significant figures
char	Single character	2	System.Char	N/A
bool	Boolean	1	System.Boolean	True or false
DateTime	Moments in time	8	System.DateTime	0:00:00 on 01/01/0001 to23:59:59 on 12/31/9999
string	Sequence of characters	2 per character	System.String	N/A

Statements

In C#, a statement is considered a command. Statements perform some action in your code such as calling a method or performing calculations. Statements are also used to declare variables and assign values to them.

Statements are formed from tokens. These tokens can be keywords, identifiers (variables), operators, and the statement terminator which is the semicolon (;). All statements in C# must be terminated with a semicolon.

Identifiers

In C#, an Identifier is name you give to the elements in your program. Elements in your program include;

- **Namespaces** - the .NET Framework uses namespaces as a way to separate class files into related buckets or categories. it also helps avoid naming collisions in applications that may contain classes with the same name
- **Classes** - classes are the blueprints for reference types. They specify the structure an object will take when you create instances of the class

- **Methods** - covered in week three, methods are discrete pieces of functionality in an application. They are analogous to functions in the non-OOP world
- **Variables** - these are identifiers that you create to hold values or references to objects in your code. A variable is essentially a named memory location

  When you create a variable in C# you must give it a data type. You can assign a value to the variable at the time you create it or later in your program code. C# will not allow you to use an unassigned variable to help prevent unwanted data from being used in your application. The following code sample demonstrates declaring a variable and assigning a value to it.

  ```
  int myVar = 0;
  ```

  C# has some restrictions around identifiers that you need to be aware of.

  First off, identifiers are case-sensitive because C# is a case-sensitive language. That means that identifiers such as myVar, _myVar, and myvar, are considered different identifiers.

  Identifiers can only contain letters (in any case), digits, and the underscore character. You can only start an identifier with a letter or an underscore character. You cannot start the identifier with a digit. **myVar** and **_myVar** are legal but **2Vars** is not.

  C# has a set of reserved keywords that the language uses. You cannot use these keywords as an identifier in your code. You may choose to take advantage of the case-sensitivity of C# and use **Double** as an identifier to distinguish it from the reserved keyword **double**, but that is not a recommended approach.

The following table contains the C# 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 (generic modifier)	int	interface internal
is	lock	long namespace	new
null	object operator	out	out (generic modifier)
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

Operators

When writing C# code, you will often use operators. An operator is a token that applies to operations on one or more operands in an expression. An expression can be part of a statement, or the entire statement. Examples include:

3 + 4 – an expression that will result in the literal value 4 being added to the literal value 3

counter++ – an expression that will result in the variable (counter) being incremented by one

Not all operators are appropriate for all data types in C#. As an example, in the preceding list the **+** operator was used to sum two numbers. You can use the same operator to combine two strings into one such as:

"Tom" + "Sawyer" which will result in a new string **TomSawyer**

You cannot use the increment operator (++) on strings however. In other words, the following example would cause an error in C#.

"Tom"++

The following table lists the C# operators by type.

Type	Operators
Arithmetic	+, -, *, /, %
Increment, decrement	++, --
Comparison	==, !=, <, >, <=, >=, is
String concatenation	+
Logical/bitwise operations	&, |, ^, !, ~, &&, ||
Indexing (counting starts from element 0)	[ ]
Casting	( ), as
Assignment	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, ??
Bit shift	<<, >>
Type information	sizeof, typeof
Delegate concatenation and removal	+, -
Overflow exception control	checked, unchecked
Indirection and Address (unsafe code only)	*, ->, [ ], &
Conditional (ternary operator)	?:

Data Conversions

C# supports two inherent types of conversion (casting) for data types, implicit and explicit. C# will use implicit conversion where it can, mostly in the case when a conversion will not result in a loss of data or when the conversion is possible with a compatible data type. The following is an example of an implicit data conversion.

Converting from smaller to larger integral types:

```
int myInt = 2147483647;
long myLong= myInt;
```

The long type has a 64-bit size in memory while the int type uses 32-bits. Therefore, the long can easily accomodate any value stored in the int type. Going from a long to an int may result in data loss however and you should use explicit casting for that.

Explicit casts are accomplished in one of two ways as demonstrated with the following coe sample.

```
double myDouble = 1234.6;
int myInt;
// Cast double to int by placing the type modifier ahead of the type to be converted
// in parentheses
myInt = (int)myDouble;
```

The second option is to use the methods provided in the .NET Framework.

```
double myDouble = 1234.6;
int myInt;
// Cast double to int by using the Convert class and the ToInt32() method.
// This converts the double value to a 32-bit signed integer
myInt = Convert.ToInt32(myDouble);
```

You will find many other methods in the Convert class that cast to different integral data types such asToBoolean(), ToByte(), ToChar(), etc.

The Convert.ToInt32() method can also be used to cast a string literal to a numeric data type. For example, you may have GUI-based application in which uses input data into text boxes. These values are string values when passed to the code in your application. Use of the above method to cast the string to numbers can help prevent exceptions in your code when trying to use the wrong data type in a specific area.

C# also provides another mechanism to deal with casting types. The use of the TryParse() method andParse() methods can help with casting as well. These methods are attached to the types in C# rather than the Convert class. An example will help demonstrate.

```
// TryParse() example
bool result = Int32.TryParse(value, out number);

// Parse() example
int number = Int32.Parse(value);
```

In the TryParse() example, the method returns a Boolean result indicating if the conversion succeeded. In the Parse() example, if the conversion does not succeed, an exception will be thrown.

Assignment

For this module, you have learned about the data types, operators, and statements in C#.

Using this knowledge you will start to form some of the foundation for your application. This assignment is designed to allow you to focus on the data types that are appropriate for attributes of real-world objects. Later in the course, you will start to combine these attributes with behaviors (methods) and then transition them into object-oriented classes.

Also, this assignment offers suggestions for attributes for each object but does not give explicit instructions on every field required. You will be required to think through the attributes of these "objects" as you create the necessary data to support them in an application. Your list may be different in an application that you would create on your own.

1. Create a C# Console application.

2. Within the Main() method in this application, create variables of the correct data type for the information related to a **student only**. The other information will be used in later modules when you create class files for the other objects listed:

Student Information

First Name	Last Name	Birthdate	Address Line 1	Address Line 2	City	State/Province	Zip/Postal	Country

Teacher Information

First Name	Last Name	Birthdate	Address Line 1	Address Line 2	City	State/Province	Zip/Postal	Country

UProgram Information

Program Name	Department Head	Degrees

Degree Information

Degree Name	Credits Required

Course Information

Course Name	Credits	Duration in Weeks	Teacher

3. Once you have the variables created, use assignment statements to assign values to one set of student variables and use the Console.WriteLine() method to output the values to the console window.

This assignment is merely intended to check your understanding of how to create variables, assign values to them, and output the information to a console window. You will build on these concepts and begin to create more functionality in later modules.

Challenge

Investigate the .NET Framework documenation around Console.ReadLine() and modify your code to use this method for accepting input from a user of your application. Using Console.ReadLine(), prompt a user for information about a student. One prompt for each student variable you created earlier. Use the appropriate code to assign the values from the user to the variables for the student.