{} Basic Concepts

Variables

Programs typically use data to perform tasks.
Creating a **variable** reserves a memory location, or a space in memory, for storing values. It is called **variable** because the information stored in that location can be changed when the program is running.
To use a variable, it must first be declared by specifying the **name** and **data type**.
A variable name, also called an **identifier**, can contain letters, numbers and the underscore character (_) and must start with a letter or underscore.
Although the name of a variable can be any set of letters and numbers, the best identifier is **descriptive** of the data it will contain. This is very important in order to create clear, understandable and readable code!

For example, **firstName** and **lastName** are good descriptive variable names, while **abc** and **xyz** are not.

Variable Types

A **data type** defines the information that can be stored in a variable, the size of needed memory and the operations that can be performed with the variable.
For example, to store an integer value (a whole number) in a variable, use the **int** keyword:

```
int myAge;
```

The code above declares a variable named **myAge** of type **integer**.

A line of code that completes an action is called a statement. Each statement in C# must end with a **semicolon**.

You can assign the value of a variable when you declare it:

```
int myAge = 18;
```

or later in your code:

```
int myAge;
myAge = 18;
```

Remember that you need to declare the variable before using it.

Built-in Data Types

There are a number of built-in data types in C#. The most common are:
int - integer.
float - floating point number.
double - double-precision version of float.
char - a single character.
bool - Boolean that can have only one of two values: True or False.
string - a sequence of characters.
The statements below use C# data types:

```
int x = 42;
double pi = 3.14;
char y = 'Z';
bool isOnline = true;
string firstName = "David";
```

Note that **char** values are assigned using single quotes and **string** values require double quotes.
You will learn how to perform different operations with variables in the upcoming lessons!

Displaying Output

Most applications require some **input** from the user and give **output** as a result.
To display text to the console window you use the **Console.Write** or **Console.WriteLine** methods.
The difference between these two is that **Console.WriteLine** is followed by a line terminator, which moves the cursor to the next line after the text output.
The program below will display Hello World! to the console window:

```
static void Main(string[] args)
{
  Console.WriteLine("Hello World!");
}
```

Try It Yourself

Note the **parentheses** after the **WriteLine** method. This is the way to pass data, or arguments, to methods. In our case **WriteLine** is the method and we pass "Hello World!" to it as an argument. String arguments must be enclosed in quotation marks.

Displaying Output

We can display variable values to the console window:

```
static void Main(string[] args)
{
  int x = 89;
  Console.WriteLine(x);
}
// Outputs 89
```

Try It Yourself

To display a **formatted string**, use the following syntax:

```
static void Main(string[] args)
{
  int x = 10;
  double y = 20;

  Console.WriteLine("x = {0}; y = {1}", x, y);
}
// Output: x = 10; y = 20
```

Try It Yourself

As you can see, the value of **x** replaced **{0}** and the value of **y** replaced **{1}**.

You can have as many variable placeholders as you need. (i.e.: {3}, {4}, etc.).

User Input

You can also prompt the user to enter data and then use the **Console.ReadLine** method to assign the input to a string variable.
The following example asks the user for a name and then displays a message that includes the input:

```
static void Main(string[] args)
{
  string yourName;
  Console.WriteLine("What is your name?");

  yourName = Console.ReadLine();

  Console.WriteLine("Hello {0}", yourName);
}
```

Try It Yourself

The **Console.ReadLine** method waits for user input and then assigns it to the variable. The next statement displays a formatted string containing Hello with the user input. For example, if you enter David, the output will be Hello David.

Note the empty parentheses in the **ReadLine** method. This means that it does not take any arguments.

User Input

The **Console.ReadLine()** method returns a **string** value.
If you are expecting another type of value (such as int or double), the entered data must be converted to that type.
This can be done using the **Convert.ToXXX** methods, where XXX is the .NET name of the type that we want to convert to. For example, methods include **Convert.ToDouble** and **Convert.ToBoolean**.
For integer conversion, there are three alternatives available based on the bit size of the integer: **Convert.ToInt16**, **Convert.ToInt32** and **Convert.ToInt64**. The default int type in C# is 32-bit.
Let's create a program that takes an integer as input and displays it in a message:

```
static void Main(string[] args)
{
  int age = Convert.ToInt32(Console.ReadLine());
  Console.WriteLine("You are {0} years old", age);
}
```

Try It Yourself

If, in the program above, a non-integer value is entered (for example, letters), the **Convert** will fail and cause an error.

Comments

Comments are explanatory statements that you can include in a program to benefit the reader of your code.
The compiler ignores everything that appears in the comment, so none of that information affects the result.

A comment beginning with two slashes (//) is called a single-line comment. The slashes tell the compiler to ignore everything that follows, until the end of the line.

```
// Prints Hello
Console.WriteLine("Hello");
```

Try It Yourself

When you run this code, Hello will be displayed to the screen. The // Prints Hello line is a comment and will not appear as output.

Multi-Line Comments

Comments that require multiple lines begin with /* and end with */ at the end of the comment block.
You can place them on the same line or insert one or more lines between them.

```
/* Some long
   comment text
*/
int x = 42;
Console.WriteLine(x);
```

Try It Yourself

Adding comments to your code is good programming practice. It facilitates a clear understanding of the code for you and for others who read it.

The var Keyword

A variable can be explicitly declared with its **type** before it is used.
Alternatively, C# provides a handy function to enable the compiler to determine the type of the variable automatically based on the expression it is assigned to.
The **var** keyword is used for those scenarios:

```
var num = 15;
```

The code above makes the compiler determine the type of the variable. Since the value assigned to the variable is an integer, the variable will be declared as an integer automatically.

The var Keyword

Variables declared using the **var** keyword are called **implicitly typed** variables.
Implicitly typed variables **must** be initialized with a value.
For example, the following program will cause an error:

```
var num;
num = 42;
```

Although it is easy and convenient to declare variables using the **var** keyword, overuse can harm the readability of your code. Best practice is to explicitly declare variables.

Constants

Constants store a value that cannot be changed from their initial assignment.
To declare a constant, use the **const** modifier.
For example:

```
const double PI = 3.14;
```

The value of const PI cannot be changed during program execution.
For example, an assignment statement later in the program will cause an error:

```
const double PI = 3.14;
PI = 8; //error
```

Constants **must** be initialized with a value when declared.

Operators

An **operator** is a symbol that performs mathematical or logical manipulations.

Arithmetic Operators
C# supports the following arithmetic operators:

Operator	Symbol	Form
Addition	+	x + y
Subtraction	-	x - y
Multiplication	*	x * y
Division	/	x / y
Modulus	%	x % y

For example:

```
int x = 10;
int y = 4;
Console.WriteLine(x-y);

//Outputs 6
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Division

The division operator (/) divides the first operand by the second. If the operands are both integers, any remainder is dropped in order to return an integer value.
Example:

```
int x = 10 / 4;
Console.WriteLine(x);

// Outputs 2
```

Try It Yourself

Division by 0 is undefined and will crash your program.

Modulus

The modulus operator (%) is informally known as the remainder operator because it returns the remainder of an integer division.
For example:

```
int x = 25 % 7;
Console.WriteLine(x);

// Outputs 4
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Operator Precedence

Operator **precedence** determines the grouping of terms in an expression, which affects how an expression is evaluated. Certain operators take higher precedence over others; for example, the multiplication operator has higher precedence than the addition operator.
For example:

```
int x = 4+3*2;
Console.WriteLine(x);
// Outputs 10
```

Try It Yourself

The program evaluates 3*2 first, and then adds the result to 4.
As in mathematics, using **parentheses** alters operator precedence.

```
int x = (4 + 3) *2;
Console.WriteLine(x);

// Outputs 14
```

Try It Yourself

The operations within parentheses are performed first. If there are parenthetical expressions nested within one another, the expression within the innermost parentheses is evaluated first.

If none of the expressions are in parentheses, multiplicative (multiplication, division, modulus) operators will be evaluated before additive (addition, subtraction) operators. Operators of equal precedence are evaluated from left to right.

Assignment Operators

The = **assignment** operator assigns the value on the right side of the operator to the variable on the left side.

C# also provides **compound assignment operators** that perform an operation and an assignment in one statement.
For example:

```
int x = 42;
x += 2; // equivalent to x = x + 2
x -= 6; // equivalent to x = x - 6
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Assignment Operators

The same shorthand syntax applies to the multiplication, division, and modulus operators.

```
x *= 8; // equivalent to x = x * 8
x /= 5; // equivalent to x = x / 5
x %= 2; // equivalent to x = x % 2
```

The same shorthand syntax applies to the multiplication, division, and modulus operators.

Increment Operator

The **increment** operator is used to increase an integer's value by one, and is a commonly used C# operator.

```
x++; //equivalent to x = x + 1
```

For example:

```
int x = 10;
x++;
Console.WriteLine(x);

//Outputs 11
```

Try It Yourself

The increment operator is used to increase an integer's value by one.

Prefix & Postfix Forms

The increment operator has two forms, **prefix** and **postfix**.

```
++x; //prefix
x++; //postfix
```

Prefix increments the value, and then proceeds with the expression.
Postfix evaluates the expression and then performs the incrementing.

Prefix example:

```
int x = 3;
int y = ++x;
// x is 4, y is 4
```

Try It Yourself

Postfix example:

```
int x = 3;
int y = x++;
// x is 4, y is 3
```

Try It Yourself

The **prefix** example increments the value of x, and then assigns it to y.
The **postfix** example assigns the value of x to y, and then increments x.

Decrement Operator

The **decrement** operator (--) works in much the same way as the increment operator, but instead of increasing the value, it decreases it by one.

```
--x; // prefix
x--; // postfix
```

The decrement operator (--) works in much the same way as the increment operator.

End.