Declaring Variables

The first step toward using a variable is to declare it. Let's look at some code.

```
// String
string myString = "There is a fox.";

// Whole number
int wholeNumber = 10;
```

Use the code above to identify the parts of a variable declaration.

- The words in blue, are called types. The type tells the computer what rules are associated with the data stored in that variable. For example, the type "string", tells the computer to expect a list of letters (typically referred to as "characters" in programming). The type "int" is short for integer, which is a whole number like 1,2,3,etc. If you assigned a number like 10 to a string variable, it would cause an error.
- The text following the type is the variable name. (i.e. myString, wholeNumber)
- The text following the name is the assignment operator. Yes, what you think of as an equals sign, is actually called an assignment operator.
- Next, we have the value. (i.e. "There is a fox.",10)
- Finally, we have the semicolon which terminates or ends the statement.

To state it in everyday terms, we can replace the equal sign with the word "gets" and read this line of code as: "myString gets the text 'There is a fox'". To talk like a programmer, you might say, "the string variable 'myString' is assigned the string value 'There is a fox'".

Another Example:

```
// Declare an integer for tracking player coins, initialize it with a value of 5
int playerCoins = 5;
```

The name of the variable is *playerCoins* and it is an integer type. The initial value is set to five by using the assignment operator. The naming of the variable is completely up to the programmer, but it should make sense according to its purpose.

If we do not provide a value when declaring the variable, it will start with a default value for it's type. For integers, the default value is 0. We use the term "initialization" to describe the first time we set a value for a variable.

```
// Declare an integer for tracking player coins. Default value will be 0.
int playerCoins;
```

There are several rules we must follow when naming a variable.

- Variable names may not contain spaces
- Variables can not share the name of a previously defined variable

- Variable names may not start with a number

```
// This variable declaration is fine
int playerCoins = 3;

// This variable declaration will cause a compiler error. Variable names must be unique
int playerCoins = 4;
```

Even though a variable must be one word, we can use numbers and some symbols to make them unique:

```
// Declare a variable named playerCoins1, set value to 2
int playerCoins1 = 2;

// Declare a variable named playerCoins2, set value to 3
int playerCoins2 = 3;

// Declare a variable named _playerCoins, set value to 4
int _playerCoins = 4;
```

The code above declares three separate integer variables, each with a unique name and its own value.

Lastly, for brevity in writing code, we can also declare multiple variables of the same type on the same line, using the keyword for the type only once:

```
// Declare an integer named x and initialize it to 10. Also declare an integer named y and
initialize it to 20
int x = 10, y = 20;
```

Rather than placing a semi-colon at the end of each declaration, we separate the variable declarations using a comma. We only need to use a semi-colon at the very end of the line.

Now that you've seen some examples of declaring a variable, let's manipulate variable values.

Variable Manipulation

As long as we know the name of the variable, we can typically read or write to it. For example, we can declare an integer for tracking the high score of a game. Once it's declared, we can manipulate its data:

```
// Declare an integer named highScore. Value is 0.
int highScore;

// Set highScore to the value of 10
highScore = 10;

// Set highScore to the value of itself plus 10. Value should be 20.
highScore = highScore + 10;
```

The above code shows a variable declaration and then changes the value of the variable twice. When changing the value of a variable, we do not need to state its type again and doing so would cause an error. Also note that the last line of code set the value of the *highScore* variable to itself plus another value.

We can also declare and set variables based on other variables:

```
int x = 3;
int y = 2;

int result = x + y;
```

The above code shows the declaration and setting of three separate variables. The first two (x and y) store values in integer form. The third (z) is the result of adding x and y together.

Code Re-factoring

Code refactoring is similar to factoring in math where we take an expression and reduce it to its simplest form. For an example, imagine we were using some code to calculate the result of the mathematical expression 4^3.

Consider the following:

```
int base1 = 4;
int base2 = 4;
int base3 = 4;
int result = base1 * base2 * base3;
```

This code will execute perfectly and will calculate an accurate result of 64. Now, let's pretend we want to change the base number from a 4 to a 7. To do this, we would need to change three instances of the number "4":

```
int base1 = 4;
int base2 = 4;
int base3 = 4;
```

Is this the most efficient way of changing the code?

Well, we wouldn't be asking the question if the answer was “yes”, so let us explain a better way. Since the values of the variables are all the same, in this case a 7, let's switch from using three variables to only one.

Now the code looks like this:

```
int baseNum = 4;
int result = baseNum * baseNum * baseNum;
```

You've done the work of four lines of code in only two. This is better programming because it increases code reuse, which in turn, can reduce the likelihood of typos since the code is more concise (less code, less opportunity for errors). Programmers are always looking for opportunities to refactor code.

In the refactored code above, we used the variable named “baseNum” multiple times (in this case, three times). Reuse of a variable is a very common practice and it's not unusual for a variable to be set at the start of a program, never changed again, and it's value used to calculate hundreds, thousands, or even millions of results. Programmers call a variable that is never intended to change a constant.

C# Built-In Types

C# shares many built-in types with other C-based languages, such as C and C++. Built-In types are provided as a standard part of the programming language.

In this topic we will cover the most common built-in types. As we mentioned earlier, types define the rules for how our computer, and more specifically our compiler, enforces rules about the use of data. And while errors sound like a negative concept, it's much better to catch an error while you are building the software than it is after the software has already shipped. Many error messages generated from a type give us a good hint about a problem we didn't want to occur. An ounce of prevention is worth a pound of cure, which is exactly what types help provide.

int

The keyword *int* is short for "integer" and is used to store whole numbers. All types will have some limits to how you can use them. For example, ints must be between -2,147,483,648 and 2,147,483,647. A few examples where an int would be an appropriate type include tracking game score, player lives, ammo count, and remaining time in a game.

```
// Declare and increase the game score
int gameScore = 0;

int negativeNumber = -78;

// Add to the gameScore
gameScore = gameScore + 100;
```

float

The keyword *float* is used to store floating-point values. A floating point number has a dot in it, such as 4.567. When specifying a value for a float we need to add a "f" after the number.

```
// Store the value of 0.5 in a float named 'y'
float y = 0.5f;

float negativeNumber = -10000.5f;
```

string

The keyword *string* is used to store to store multiple characters (e.g. letters). When assigning characters to a string in C#, we must use double quotation marks ("").

```
string myString = "Hello";

string myOtherString = "World";
```

We can also combine multiple strings together using the plus operator (+), which is known as concatenation:

```
string firstString = "Hello";
string secondString = "World";

// Concatenating firstString and secondString. Result will be "HelloWorld"
string thirdString = firstString + secondString;

// Replace the value of thirdString with firstString, the string literal of a space, the secondString,
and the string literal of an exclamation point
// Result will be "Hello World!"
thirdString = firstString + " " + secondString + "!";
```

Additionally, we can convert values of a non-string type to string data:

```
int myInteger = 42;

// convertedString will contain the value "42"
string convertedString = myInteger.ToString();
```

bool

The *bool* keyword is used to represent Boolean values. A bool can only contain one of two values; true or false. In short, a bool can be used in programming to control which blocks of code will execute.

```
// Declare a boolean named myBoolean
bool myBoolean = true;
```

The value of a boolean can be changed at any time. The data is very useful when programming code that can toggle on and off. A real world example would be the state of a light switch.

```
bool lightIsOn = true;
```

uint

The keyword *uint* is short for "unsigned integer" and is used to store positive, whole numbers. Unsigned shorts have a range of 0 to 4,294,967,295. If we know we want data to always remain positive and increase in size up to a large value, a uint is a better choice than int. For example, a variable tracking the number of player lives should never be less than zero:

```
// Store the value of 50 in an unsigned integer named 'ammo'
uint ammo = 50U;
```

```
// Change ammo's value to the result of its own value - 100. This will result in an error, since uint cannot be less than 0
ammo = ammo - 100U;
```

double

The keyword *double* is like a float but larger. I bet you are wondering why there are floats and doubles? The reason is that doubles use more computer resources as a trade off for better accuracy.

```
// Store the value of 0.5 in a double named 'x'
double x = 0.5;
```

Types & Negative Values

When a type can have negative values, it is referred to as **signed**. If it can only contain positive values, it is **unsigned**.:

```
// Signed int can be negative
int mySignedInt = -5;

// Unsigned cannot be negative. This code will cause a compile error.
uint myUnsignedInt = -10;
```

Suffixes - A Syntax Gotcha

Be aware that sometimes you will need to use a suffix to help the compiler know what type you are trying to assign to a variable. Floats are the most common case where a suffix is required:

```
// Create a double with a value of 0.5
double myDouble = 0.5;

// Create a decimal with a value of 0.5
decimal myDecimal = 0.5m;

// Create a float with a value of 0.5
float myFloat = 0.5f;

// The compiler thinks a double is trying to store in a float, which is a type error.
float otherFloat = 0.5;
```

Type Casting

Sometimes we need to convert a value from one type to another. For example, you may want to display the integer number "2" to display how many player lives remain. In that case, we would need to convert the "2" to a string. This is known as type casting. Each programming language has its own rules and features for type casting. In C#, type casting is either implicit or explicit. Let's take a look at an explicit cast.

```
// Explicity type cast of a float to an integer.
float myFloat = 3.4f;
int myInt = (int)myFloat;//myInt will truncate (i.e. remove values after the decimal point) to a
value of 3;
```

We wrap the type, in this case int, with parenthesis and place it to the left of the expression we want to cast.

let's look at an example of an implicit cast:

```
// Store the maximum value of an int
int maxInt = 2147483647;

// Store the value of maxInt in a long, which is a larger version of an int
long myLong = maxInt;
```

Notice how we don't need to include the type and parenthesis as we did in the previous example. The implicit cast occurred automatically since casting from an int to a long had no adverse effect, such as reducing the number's fidelity.

Let's look at another example of an explicit cast. A common example is converting between integer values and floating point values.

```
// Store the floating point value of 3.4 in myFloat
float myFloat = 3.4f;

// Explicitly convert myFloat to the integer myInt. The value of myInt will be 3
int myInt = (int)myFloat;
```

In the above code example, the syntax for explicit conversion required (int) before the assignment (=). Some fidelity is lost in this example since the numbers after the floating-point will be removed, or truncated by the cast.

Entire operations can be cast, if desired:

```
// Take the integral sum of 3 + 4 + 5 + 6 (18) and store it in myFloat, resulting in 18.0f
float myFloat = (float)(3 + 4 + 5 + 6);
```

Because of implicit casting, some seemingly different types can work together. An example is storing numeric values in a string. This can be done automatically using the plus symbol. This is useful for situations like printing a game's score to the screen with additional text:

```
// Set an integer for high score to 100
int highScore = 100;

// Create a string to draw on screen that says "High Score" and concatenate the actual score
string highScoreText = "High Score: " + highScore;
```

In the above code, we do not have to use any explicit casting. *highScoreText* will automatically convert the numeric value of the high score to a string.