Nullable Types

C# supports the concept of nullable data types. Simply put, a nullable type can represent all the values of its underlying type, plus the value null. Thus, if you declare a nullable bool, it could be assigned a value from the set {true, false, null}. This can be extremely helpful when working with relational databases, given that it is quite common to encounter undefined columns in database tables. Without the concept of a nullable data type, there is no convenient manner in C# to represent a numerical data point with no value. ***Value types can never be assigned the value of null, as that is used to establish an empty object reference.***

To define ***a nullable variable type****, the question mark symbol (*__?__*) is suffixed to the underlying data type. Do note that this syntax is legal only when applied to value types.* If you attempt to create a nullable reference type (including strings), you are issued a compile-time error. Like a non-nullable variable, local nullable variables must be assigned an initial value before you can use them.

```
// Define some nullable variables.
int? nullableInt = 10;
double? nullableDouble = 3.14;
bool? nullableBool = null;
char? nullableChar = 'a';
int?[] arrayOfNullableInts = new int?[10];

// Error! Strings are reference types!
// string? s = "oops";
```

In C#, the ? suffix notation is a shorthand for creating an instance of the generic System.Nullable<T> structure type. It is important to understand that the System.Nullable<T> type provides a set of members that all nullable types can make use of. So, we can implement our variables like this:

```
// Define some local nullable types using Nullable<T>.
Nullable<int> nullableInt = 10;
Nullable<double> nullableDouble = 3.14;
Nullable<bool> nullableBool = null;
Nullable<char> nullableChar = 'a';
Nullable<int>[] arrayOfNullableInts = new Nullable<int>[10];

Console.WriteLine(nullableBool.HasValue);    // False
Console.WriteLine(nullableBool.Value);    // System.InvalidOperationException: Nullable
object must have a value.
```

The Null Coalescing Operator (??)

Any variable that might have a null value (i.e., a reference-type variable or a nullable value-type variable) can make use of the C# ?? operator, which is formally termed the null coalescing operator. This operator allows you to assign a value to a nullable type if the retrieved value is in fact null. The benefit of using the ?? operator is that it provides a more compact version of a traditional if/else condition.

```
int? variable1 = null;
int? variable2 = 2;

if (variable1 == null)
{
        variable2 = 5;
}
else
{
        variable2 = variable1;
}
Console.WriteLine(variable2);   //  5

variable2 = variable1 ?? 5;
Console.WriteLine(variable2);   // 5
```

Tuples

Tuples use the ValueTuple data type, potentially saving significant memory. The ValueTuple data type creates different structs based on the number of properties for a tuple.

```
(string, int, string) values = ("a", 5, "c");
```

By default, the compiler assigns each property the name ItemX, where X represents the one based position in the tuple.

```
Console.WriteLine($"First item: {values.Item1}");
Console.WriteLine($"Second item: {values.Item2}");
Console.WriteLine($"Third item: {values.Item3}");
```

Specific names can also be added to each property in the tuple on either the right side or the left side of the statement.

```
(string FirstLetter, int TheNumber, string SecondLetter) valuesWithNames = ("a", 5, "c");
var valuesWithNames2 = (FirstLetter: "a", TheNumber: 5, SecondLetter: "c");

Console.WriteLine($"First item: {valuesWithNames.FirstLetter}");     // a
Console.WriteLine($"Second item: {valuesWithNames.TheNumber}");    // 5
Console.WriteLine($"Third item: {valuesWithNames.SecondLetter}");    // c

//Using the item notation still works!
Console.WriteLine($"First item: {valuesWithNames.Item1}");    // a
Console.WriteLine($"Second item: {valuesWithNames.Item2}");    // 5
Console.WriteLine($"Third item: {valuesWithNames.Item3}");  // c
```