# C# Basic Data Type Classes

Because C# is statically-typed at compile time, after a variable is declared, it cannot be declared again or used to store values of another type unless that type is convertible to the variable's type. For example, there is no conversion from an integer to any arbitrary string. Therefore, after you declare i as an integer, you cannot assign the string "Hello" to it, as is shown in the following code.

***int i;***

***i = "Hello"; // Error: "Cannot implicitly convert type string to int"***

However, you might sometimes need to copy a value into a variable or method parameter of another type. For example, you might have an integer variable that you need to pass to a method whose parameter is typed as double. Or you might need to assign a class variable to a variable of an interface type. These kinds of operations are called type conversions. In C#, you can perform the following kinds of conversions:

**Implicit conversions**

No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.

**Explicit conversions**

Explicit conversions require a cast operator. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.

**User-defined conversions**

User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship.

**Conversions with helper classes**

To convert between non-compatible types, such as integers and System.DateTime objects, or hexadecimal strings and byte arrays, you can use the System.BitConverter class, the System.Convert class, and the Parse methods of the built-in numeric types, such as Int32.Parse.

# C# Base Data Type Classes

C++ was very rich in datatypes, but that leads to confusion too. Especially, when you write components that may be consumed by applications written in other platforms, you have to make sure the types used are compatible with other platforms too! .NET types start from a clean slate. All .NET languages share the same types. So, they are all compatible and no worries.This means, you can call C# code from VB.NET and vice versa, without worrying about type conversions.The following list shows the list of data types available in C# and their corresponding class/struct in .NET class library.

|  |  |
| --- | --- |
| C# Data type | Mapped to .NET class/struct |
| sbyte | System.SByte |
| byte | System.Byte |
| char | System.Char |
| float | System.Single |
| decimal | System.Decimal |
| double | System.Double |
| ushort | System.UInt16 |
| short | System.Int16 |
| uint | System.UInt32 |
| int | System.Int32 |
| ulong | System.UInt64 |
| long | System.Int64 |
| bool | System.Boolean |
| string | System.String |
| object | System.Object |

# Value Types & Reference Types

In C# data types are classified into two :

• value types

• reference types

|  |  |
| --- | --- |
| value types | reference types |
| allocated on stack | allocated on heap |
| a value type variable contains the data itself | reference type variable contains the address of memory location where data is actually stored. |
| when you copy a value type variable to another one, the actual data is copied and each variable can be independently manipulated. | when copying a reference type variable to another variable, only the memory address is copied. Both variables will still point to thesame memory location, which means, if you change one variable, the value will be changed for the other variable too. |
| integer, float, boolean, double etc are value types. | string and object are reference types. |
| struct is value type. | classes and interfaces are reference types. |

# Convert Class - Example Of Helper Class

This class returns a type whose value is equivalent to the value of a specified type. The supported base types are Boolean, Char, SByte, Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Decimal, DateTime and String. A conversion method exists to convert every base type to every other base type. However, the actual conversion operation performed falls into three categories:

* A conversion from a type to itself simply returns that type. No conversion is actually performed.
* A conversion which cannot yield a meaningful result throws InvalidCastException. No conversion is actually performed. An exception is thrown for conversions from Char to Boolean, Single, Double, Decimal or DateTime, and from those types to Char. An exception is thrown for conversions from DateTime to any type except String, and from any type, except String, to DateTime.
* Any base type, other than those described above, can be converted to and from any other base type.

An exception will not be thrown if the conversion of a numeric type results in a loss of precision (that is, the loss of some least significant digits). However, an exception will be thrown if the result is larger than can be represented by the particular conversion method's return value type.

For example, when a Double is converted to a Single, a loss of precision might occur but no exception is thrown. However, if the magnitude of the Double is too large to be represented by a Single, an overflow exception is thrown.

# Boxing and Unboxing

Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type. When the CLR boxes a value type, it wraps the value inside a System.Object and stores it on the managed heap. Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system, in which a value of any type can be treated as an object.

In the following example, the integer variable i is boxed and assigned to object o.

int i = 123;

// The following line boxes i.

object o = i;

The object o can then be unboxed and assigned to integer variable i:

o = 123;

i = (int)o; // unboxing