**The variables in C#, are categorized into the following types −**

* Value types
* Reference types
* Pointer types

Value Type

Value type variables can be assigned a value directly. They are derived from the class **System.ValueType**.

The value types directly contain data. Some examples are **int, char, and float**, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an **int** type, the system allocates memory to store the value.

The following table lists the available value types in C# 2010 −

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Represents** | **Range** | **Default Value** |
| bool | Boolean value | True or False | False |
| byte | 8-bit unsigned integer | 0 to 255 | 0 |
| char | 16-bit Unicode character | U +0000 to U +ffff | '\0' |
| decimal | 128-bit precise decimal values with 28-29 significant digits | (-7.9 x 1028 to 7.9 x 1028) / 100to 28 | 0.0M |
| double | 64-bit double-precision floating point type | (+/-)5.0 x 10-324 to (+/-)1.7 x 10308 | 0.0D |
| float | 32-bit single-precision floating point type | -3.4 x 1038 to + 3.4 x 1038 | 0.0F |
| int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0 |
| long | 64-bit signed integer type | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L |
| sbyte | 8-bit signed integer type | -128 to 127 | 0 |
| short | 16-bit signed integer type | -32,768 to 32,767 | 0 |
| uint | 32-bit unsigned integer type | 0 to 4,294,967,295 | 0 |
| ulong | 64-bit unsigned integer type | 0 to 18,446,744,073,709,551,615 | 0 |
| ushort | 16-bit unsigned integer type | 0 to 65,535 | 0 |

# C# - Type Conversion

Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms −

* **Implicit type conversion** − These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.
* **Explicit type conversion** − These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

using System;

namespace TypeConversionApplication {

class ExplicitConversion {

static void Main(string[] args) {

double d = 5673.74;

int i;

// cast double to int.

i = (int)d;

Console.WriteLine(i);

Console.ReadKey();

}

}

}

## C# Type Conversion Methods

C# provides the following built-in type conversion methods −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **ToBoolean**  Converts a type to a Boolean value, where possible. |
| 2 | **ToByte**  Converts a type to a byte. |
| 3 | **ToChar**  Converts a type to a single Unicode character, where possible. |
| 4 | **ToDateTime**  Converts a type (integer or string type) to date-time structures. |
| 5 | **ToDecimal**  Converts a floating point or integer type to a decimal type. |
| 6 | **ToDouble**  Converts a type to a double type. |
| 7 | **ToInt16**  Converts a type to a 16-bit integer. |
| 8 | **ToInt32**  Converts a type to a 32-bit integer. |
| 9 | **ToInt64**  Converts a type to a 64-bit integer. |
| 10 | **ToSbyte**  Converts a type to a signed byte type. |
| 11 | **ToSingle**  Converts a type to a small floating point number. |
| 12 | **ToString**  Converts a type to a string. |
| 13 | **ToType**  Converts a type to a specified type. |
| 14 | **ToUInt16**  Converts a type to an unsigned int type. |
| 15 | **ToUInt32**  Converts a type to an unsigned long type. |
| 16 | **ToUInt64**  Converts a type to an unsigned big integer. |

# C# - Nullables

C# provides a special data types, the **nullable** types, to which you can assign normal range of values as well as null values.

For example, you can store any value from -2,147,483,648 to 2,147,483,647 or null in a Nullable<Int32> variable. Similarly, you can assign true, false, or null in a Nullable<bool> variable. Syntax for declaring a **nullable** type is as follows −

< data\_type> ? <variable\_name> = null;

The following example demonstrates use of nullable data types −

using System;

namespace CalculatorApplication {

class NullablesAtShow {

static void Main(string[] args) {

int? num1 = null;

int? num2 = 45;

double? num3 = new double?();

double? num4 = 3.14157;

bool? boolval = new bool?();

// display the values

Console.WriteLine("Nullables at Show: {0}, {1}, {2}, {3}", num1, num2, num3, num4);

Console.WriteLine("A Nullable boolean value: {0}", boolval);

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result −

Nullables at Show: , 45, , 3.14157

A Nullable boolean value:

## The Null Coalescing Operator (??)

The null coalescing operator is used with the nullable value types and reference types. It is used for converting an operand to the type of another nullable (or not) value type operand, where an implicit conversion is possible.

If the value of the first operand is null, then the operator returns the value of the second operand, otherwise it returns the value of the first operand. The following example explains this −

using System;

namespace CalculatorApplication {

class NullablesAtShow {

static void Main(string[] args) {

double? num1 = null;

double? num2 = 3.14157;

double num3;

num3 = num1 ?? 5.34;

Console.WriteLine(" Value of num3: {0}", num3);

num3 = num2 ?? 5.34;

Console.WriteLine(" Value of num3: {0}", num3);

Console.ReadLine();

}

}

}

# C# - Structures

## Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program.

For example, here is the way you can declare the Book structure −

struct Books {

public string title;

public string author;

public string subject;

public int book\_id;

};

using System;

struct Books {

public string title;

public string author;

public string subject;

public int book\_id;

};

public class testStructure {

public static void Main(string[] args) {

Books Book1; /\* Declare Book1 of type Book \*/

Books Book2; /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/

Book1.title = "C Programming";

Book1.author = "Nuha Ali";

Book1.subject = "C Programming Tutorial";

Book1.book\_id = 6495407;

/\* book 2 specification \*/

Book2.title = "Telecom Billing";

Book2.author = "Zara Ali";

Book2.subject = "Telecom Billing Tutorial";

Book2.book\_id = 6495700;

/\* print Book1 info \*/

Console.WriteLine( "Book 1 title : {0}", Book1.title);

Console.WriteLine("Book 1 author : {0}", Book1.author);

Console.WriteLine("Book 1 subject : {0}", Book1.subject);

Console.WriteLine("Book 1 book\_id :{0}", Book1.book\_id);

/\* print Book2 info \*/

Console.WriteLine("Book 2 title : {0}", Book2.title);

Console.WriteLine("Book 2 author : {0}", Book2.author);

Console.WriteLine("Book 2 subject : {0}", Book2.subject);

Console.WriteLine("Book 2 book\_id : {0}", Book2.book\_id);

Console.ReadKey();

}

}

Features of C# Structures

You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features −

* Structures can have methods, fields, indexers, properties, operator methods, and events.
* Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and cannot be changed.
* Unlike classes, structures cannot inherit other structures or classes.
* Structures cannot be used as a base for other structures or classes.
* A structure can implement one or more interfaces.
* Structure members cannot be specified as abstract, virtual, or protected.
* When you create a struct object using the **New** operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
* If the New operator is not used, the fields remain unassigned and the object cannot be used until all the fields are initialized.

Class versus Structure

Classes and Structures have the following basic differences −

* classes are reference types and structs are value types
* structures do not support inheritance
* structures cannot have default constructor

using System;

struct Books {

private string title;

private string author;

private string subject;

private int book\_id;

public void getValues(string t, string a, string s, int id) {

title = t;

author = a;

subject = s;

book\_id = id;

}

public void display() {

Console.WriteLine("Title : {0}", title);

Console.WriteLine("Author : {0}", author);

Console.WriteLine("Subject : {0}", subject);

Console.WriteLine("Book\_id :{0}", book\_id);

}

};

public class testStructure {

public static void Main(string[] args) {

Books Book1 = new Books(); /\* Declare Book1 of type Book \*/

Books Book2 = new Books(); /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/

Book1.getValues("C Programming",

"Nuha Ali", "C Programming Tutorial",6495407);

/\* book 2 specification \*/

Book2.getValues("Telecom Billing",

"Zara Ali", "Telecom Billing Tutorial", 6495700);

/\* print Book1 info \*/

Book1.display();

/\* print Book2 info \*/

Book2.display();

Console.ReadKey();

}

}

# C# - Enums

An enumeration is a set of named integer constants. An enumerated type is declared using the **enum** keyword.

C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

## Declaring *enum* Variable

The general syntax for declaring an enumeration is −

enum <enum\_name> {

enumeration list

};

Where,

* The *enum\_name* specifies the enumeration type name.
* The *enumeration list* is a comma-separated list of identifiers.

Each of the symbols in the enumeration list stands for an integer value, one greater than the symbol that precedes it. By default, the value of the first enumeration symbol is 0. For example −

enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

using System;

namespace EnumApplication {

class EnumProgram {

enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

static void Main(string[] args) {

int WeekdayStart = (int)Days.Mon;

int WeekdayEnd = (int)Days.Fri;

Console.WriteLine("Monday: {0}", WeekdayStart);

Console.WriteLine("Friday: {0}", WeekdayEnd);

Console.ReadKey();

}

}

}

# C# - Exception Handling

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

* **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
* **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
* **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
* **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

Syntax

Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

try {

// statements causing exception

} catch( ExceptionName e1 ) {

// error handling code

} catch( ExceptionName e2 ) {

// error handling code

} catch( ExceptionName eN ) {

// error handling code

} finally {

// statements to be executed

}

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Exception Classes in C#

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.

The **System.ApplicationException** class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.

The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class −

|  |  |
| --- | --- |
| **Sr.No.** | **Exception Class & Description** |
| 1 | **System.IO.IOException**  Handles I/O errors. |
| 2 | **System.IndexOutOfRangeException**  Handles errors generated when a method refers to an array index out of range. |
| 3 | **System.ArrayTypeMismatchException**  Handles errors generated when type is mismatched with the array type. |
| 4 | **System.NullReferenceException**  Handles errors generated from referencing a null object. |
| 5 | **System.DivideByZeroException**  Handles errors generated from dividing a dividend with zero. |
| 6 | **System.InvalidCastException**  Handles errors generated during typecasting. |
| 7 | **System.OutOfMemoryException**  Handles errors generated from insufficient free memory. |
| 8 | **System.StackOverflowException**  Handles errors generated from stack overflow. |