How we can convert from reference to value in a safe code:

Value Type and Reference, both are types in C# −

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. When you declare an int type, the system allocates memory to store the value.

Value Type variables are stored in the stack.

Examples are int, char, and float, which stores numbers, alphabets, and floating-point numbers, respectively.

Reference Type

It refers to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value.

Reference Type variables are stored in the heap.

Example of built-in reference types are :

object

dynamic

string

example:

string numberString = "456";

if (int.TryParse(numberString, out int result)) // Using TryParse for safe conversion

{

// Conversion successful

Console.WriteLine($"Converted value: {result}");

}

else

{

// Conversion failed

Console.WriteLine("Invalid input for conversion to int.");

}

These methods (TryParse, Parse) provide safer ways to convert from a string to a value type, as they can handle invalid input gracefully without throwing exceptions.

For custom classes or types that you define, you can implement conversion methods or operators to convert between reference and value types in a safe manner.

What are the types of Casting:

Type casting is when you assign a value of one data type to another type.

In C#, there are two types of casting:

Implicit Casting (automatically) - converting a smaller type to a larger type size

char -> int -> long -> float -> double

example:

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

Console.WriteLine(myInt); // Outputs 9

Console.WriteLine(myDouble); // Outputs 9

Explicit Casting (manually) - converting a larger type to a smaller size type

double -> float -> long -> int -> char

example:

double myDouble = 9.78;

int myInt = (int) myDouble; // Manual casting: double to int

Console.WriteLine(myDouble); // Outputs 9.78

Console.WriteLine(myInt); // Outputs 9

Type Conversion Methods

It is also possible to convert data types explicitly by using built-in methods, such as Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long):

Example

int myInt = 10;

double myDouble = 5.25;

bool myBool = true;

Console.WriteLine(Convert.ToString(myInt)); // convert int to string

Console.WriteLine(Convert.ToDouble(myInt)); // convert int to double

Console.WriteLine(Convert.ToInt32(myDouble)); // convert double to int

Console.WriteLine(Convert.ToString(myBool)); // convert bool to string

Casting- Boxing

Boxing: It refers to the process of converting a value type (such as an int, float, struct, etc.) to a reference type (such as object). When you box a value type, a new object is created on the heap, and the value of the value type is copied into that object Thus, the boxed datatype provides multiple functions to operate the primitive value. Among them, we can cite the conversion of the value to a string. So we can print it on the screen.

Advantages

Many compilers execute autoboxing as an optimization measure. So we don’t have to explicitly convert conversion a primitive datatype to its equivalent wrapper type. In such a way, we not only reduce our code footprint but also reduce the complexity.

Furthermore, boxed values have additional metadata and structural organizations that our primitive datatypes don’t have: since they are objects, they can have several fields, methods, and behaviors.

Example of casting -Boxing

int num = 10; // Value type

object obj = num; // Boxing: converting 'num' to a reference type 'obj'

Casting-Unboxing

Unboxing is the reverse transformation of boxing. So, we extract the primitive value of an object from its wrapper object.

Example:

int num = 0;

object obj = 10; // Boxed 'int' value

if (obj is int) // Checking if the boxed object is of type 'int'

{

num = (int)obj; // Unboxing: converting 'obj' back to 'int'

}

We can conclude that boxing and unboxing can make our programs slower and more memory intensive when compared to using native datatypes. But they give us additional capabilities and make our code more robust and flexible.