Casting and Type Conversion

.NET CORE

*C# is <u>statically typed</u> at compile time. After a variable is declared, it cannot be declared again or assigned a value of another type <u>unless</u> that type is implicitly convertible to the variable's type. Converting one type to another is called **type conversion**.*

Casting and Type Conversion

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions

There are 2 types of conversions in C#:

- Implicit conversions: No special syntax. Type safe. No data loss.

- Explicit conversions (casts): Explicit conversions require the cast operator (). A cast is required when data might be lost in the conversion, or when failure could occur.

Implicit Conversion

Implicit conversion is possible in:

- *numeric* types when the value to be stored can fit into the variable memory without being truncated.
- *integral* types when the <u>range</u> of the source *type* is at least as big as the target *type*.

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

Implicit conversion is always possible in *reference* types.
- When a class is converted to any one of its direct or indirect *base* classes or *interfaces*.
- No special syntax is necessary.
- Derived classes always contain all the members of the base class.

```
Derived d = new Derived();
Base b = d; // Always OK.
```

Explicit Conversion

If there is a risk of losing information, you <u>must</u> perform a *Cast*.

Specify the target *type* in *( )* in front of the value or variable to be converted.

An explicit **cast** is <u>required</u> if you need to convert from a **base** type to a *derived* type.

==*This doesn't prevent the loss of data.==

```csharp
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

```csharp
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe) a;
```

Type conversion exceptions at run time

In some *reference* type conversions, It is possible for a *cast* operation that compiles correctly to fail at run time.

A *type cast* that fails at run time will cause an **InvalidCastException** to be thrown.

```csharp
using System;

class Animal
{

    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // Cause InvalidCastException at run time
        // because Mammal is not convertible to Reptile.
        Reptile r = (Reptile)a;

    }
}
```

Type-testing and cast operators – '*is*' operator

The *is* operator checks if the runtime *type* of an expression result is compatible with a given *type*.

`E is T` returns *true* if E is non-null and can be converted to *type* T by a *reference*, a *boxing*, or an *unboxing* conversion.

```csharp
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base);    // output: True
        Console.WriteLine(b is Derived); // output: False

        object d = new Derived();
        Console.WriteLine(d is Base);    // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

Type-testing and cast operators – *'is'* operator

The *is* operator takes into account *boxing* and *unboxing* conversions but doesn't consider numeric conversions.

```csharp
int i = 27;
Console.WriteLine(i is System.IFormattable);  // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int);  // output: True
Console.WriteLine(iBoxed is long);  // output: False
```

Type-testing and cast operators – 'as' operator

The *as* operator explicitly converts the result of an expression to a given reference or nullable value type. If the conversion is not possible, the *as* operator returns null. Unlike the *cast* operator *()*, the *as* operator never throws an exception.

`E as T`   produces the same result as   `E is T ? (T)(E) : (T)null`

The *as* operator considers only *reference*, *nullable*, *boxing*, and *unboxing* conversions.

You cannot use the *as* operator to perform a user-defined conversion. To do that, use the *cast* operator *()*.

typeof

The *typeof* operator obtains the *System.Type* instance type. The argument to the *typeof* operator must be the name of a *type* or a *type parameter*.

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

typeof Operator

- An expression cannot be an argument of the *typeof* operator. To get the System.Type instance for the runtime *type* of an expression result, use *Object.GetType()*.

- Use the *typeof* operator to check if the runtime *type* of the expression result exactly matches a given *type*.

This example demonstrates the difference between type checking performed with the *typeof* operator and the *is* operator.

```csharp
public class Animal { }

public class Giraffe : Animal { }

public static class TypeOfExample
{
    public static void Main()
    {
        object b = new Giraffe();
        Console.WriteLine(b is Animal);  // output: True
        Console.WriteLine(b.GetType() == typeof(Animal));  // output: False

        Console.WriteLine(b is Giraffe);  // output: True
        Console.WriteLine(b.GetType() == typeof(Giraffe));  // output: True
    }
}
```