

# Casting and type conversions (C# Programming Guide)

Article • 01/12/2022 • 5 minutes to read • [18 contributors](#)



## In this article

[Implicit conversions](#)


[Explicit conversions](#)

[Type conversion exceptions at run time](#)

[C# language specification](#)

[See also](#)

Because C# is statically-typed at compile time, after a variable is declared, it cannot be declared again or assigned a value of another type unless that type is implicitly convertible to the variable's type. For example, the `string` cannot be implicitly converted to `int`. Therefore, after you declare `i` as an `int`, you cannot assign the string "Hello" to it, as the following code shows:

C#	 Copy
<pre>int i;  // error CS0029: Cannot implicitly convert type 'string' to 'int' i = "Hello";</pre>	

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 always succeeds 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 (casts):** Explicit conversions require a [cast expression](#). 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. For more information, see [User-defined conversion operators](#).
- **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](#). For more information, see [How to convert a byte array to an int](#), [How to convert a string to a number](#), and [How to convert between hexadecimal strings and numeric types](#).


## Implicit conversions

For built-in numeric types, an implicit conversion can be made when the value to be stored can fit into the variable without being truncated or rounded off. For integral types, this means the range of the source type is a proper subset of the range for the target type. For example, a variable of type [long](#) (64-bit integer) can store any value that an [int](#) (32-bit integer) can store. In the following example, the compiler implicitly converts the value of `num` on the right to a type `long` before assigning it to `bigNum`.

C#	 Copy
<pre>// Implicit conversion. A long can // hold any value an int can hold, and more! int num = 2147483647; long bigNum = num;</pre>	

For a complete list of all implicit numeric conversions, see the [Implicit numeric conversions](#) section of the [Built-in numeric conversions](#) article.

For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes or interfaces. No special syntax is necessary because a derived class always contains all the members of a base class.

C#	 Copy
<pre>Derived d = new Derived();</pre>	

```
// Always OK.  
Base b = d;
```

## Explicit conversions

However, if a conversion cannot be made without a risk of losing information, the compiler requires that you perform an explicit conversion, which is called a *cast*. A cast is a way of explicitly informing the compiler that you intend to make the conversion and that you are aware that data loss might occur, or the cast may fail at run time. To perform a cast, specify the type that you are casting to in parentheses in front of the value or variable to be converted. The following program casts a [double](#) to an [int](#). The program will not compile without the cast.

C#

 Copy

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

For a complete list of supported explicit numeric conversions, see the [Explicit numeric conversions](#) section of the [Built-in numeric conversions](#) article.

For reference types, an explicit cast is required if you need to convert from a base type to a derived type:

C#


 Copy

```
// 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;
```

A cast operation between reference types does not change the run-time type of the underlying object; it only changes the type of the value that is being used as a reference to that object. For more information, see [Polymorphism](#).

## Type conversion exceptions at run time

In some reference type conversions, the compiler cannot determine whether a cast will be valid. It is possible for a cast operation that compiles correctly to fail at run time. As shown in the following example, a type cast that fails at run time will cause an [InvalidCastException](#) to be thrown.

C#	 Copy
<pre>class Animal {     public void Eat() =&gt; System.Console.WriteLine("Eating.");      public override string ToString() =&gt; "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.         System.Console.WriteLine("Press any key to exit.");         System.Console.ReadKey();     }      static void Test(Animal a)     {         // System.InvalidCastException at run time         // Unable to cast object of type 'Mammal' to type 'Reptile'         Reptile r = (Reptile)a;     } }</pre>	

The Test method has an Animal parameter, thus explicitly casting the argument a to a Reptile makes a dangerous assumption. It is safer to not make assumptions, but rather check the type. C# provides the [is](#) operator to enable you to test for compatibility

before actually performing a cast. For more information, see [How to safely cast using pattern matching and the as and is operators](#).

## C# language specification

For more information, see the [Conversions](#) section of the [C# language specification](#).

## See also

- [C# Programming Guide](#)
- [Types](#)
- [Cast expression](#)
- [User-defined conversion operators](#)
- [Generalized Type Conversion](#)
- [How to convert a string to a number](#)

## Recommended content

### [Arithmetic operators - C# reference](#)

Learn about C# operators that perform multiplication, division, remainder, addition, and subtraction operations with numeric types.

### [C# operators and expressions - C# reference](#)

Learn about C# operators and expressions, operator precedence, and operator associativity.

### [+ and += operators - C# reference](#)

Learn about the C# addition operator and how it works with operands of numeric, string, or delegate types.

### [Passing arrays as arguments - C# Programming Guide](#)

Arrays in C# can be passed as arguments to method parameters. Because arrays are reference types, the method can change the value of the elements.

### [override modifier - C# Reference](#)

[override modifier - C# Reference](#)

### [Floating-point numeric types - C# reference](#)

Learn about the built-in C# floating-point types: float, double, and decimal

### [Boolean logical operators - C# reference](#)

Learn about C# operators that perform logical negation, conjunction (AND), and inclusive and exclusive disjunction (OR) operations with Boolean operands.

### [Multidimensional Arrays - C# Programming Guide](#)

Arrays in C# can have more than one dimension. This example declaration creates a two-dimensional array of four rows and two columns.

---

Show more 