**Math.Sqrt** computes a square root value at runtime. A square root is the number that, when multiplied by itself, equals the original number. Sqrt is a slower computation. It can be cached for a performance boost.

**Example.**The Math.Sqrt method is called with a double argument. In this program, the integer arguments to Math.Sqrt are converted to doubles at compile-time. The Sqrt method returns another double.[**Double**](https://www.dotnetperls.com/double)

**Based on:** .NET (2019)

**C# program that uses Math.Sqrt**

using System;

class Program

{

static double[] \_lookup = new double[5];

static void Main()

{

// Compute square roots by calling Math.Sqrt.

double a = **Math.Sqrt**(1);

double b = **Math.Sqrt**(2);

double c = **Math.Sqrt**(3);

double d = **Math.Sqrt**(4);

// Store square roots in lookup table.

var lookup = \_lookup;

lookup[1] = a;

lookup[2] = b;

lookup[3] = c;

lookup[4] = d;

Console.WriteLine(a);

Console.WriteLine(b);

Console.WriteLine(c);

Console.WriteLine(d);

Console.WriteLine(lookup[1]);

Console.WriteLine(lookup[2]);

Console.WriteLine(lookup[3]);

Console.WriteLine(lookup[4]);

}

}

**Output**

1

1.4142135623731

1.73205080756888

2

1

1.4142135623731

1.73205080756888

2

**Double** is an 8-byte numeric type. It is used to store large and small values. It also stores fractional values such as 1.5 and negative values such as -1.5. It requires more memory than an int. [**Single, Double**](https://www.dotnetperls.com/single-double)

**Example.**A double is declared in the same way as an int. You use the double type in the declaration, and can assign it using the assignment operator "=". It offers fractional values. It accommodates large and small numbers.

**Encoding:** Its encoding uses 8 bytes, twice the number of bytes in an int. The additional 4 bytes allow more representations in the type.

[**Int**](https://www.dotnetperls.com/int)

**Next:** We show that the double is aliased (mapped) to the System. Double struct—the two types are equivalent.

[**Struct**](https://www.dotnetperls.com/struct)

**MaxValue:** The maximum value of a double is expressed in scientific notation—after E we have the power of 308.

**Based on:** .NET (2019)

**C# program that uses double type**

using System;

class Program

{

static void Main()

{

// Use double type.

**double** number = 1.5;

Console.WriteLine(number);

number = -1; // Can be negative

Console.WriteLine(number);

Console.WriteLine(number == -1); // Can use == operator

Console.WriteLine(number + 100); // Can use + operator

Console.WriteLine(number.GetType());

Console.WriteLine(typeof(double));

Console.WriteLine(double.MinValue);

Console.WriteLine(double.MaxValue);

// Find the memory usage for a double value.

long bytes1 = GC.GetTotalMemory(false);

double[] array = new double[1000 \* 1000];

array[0] = 1;

long bytes2 = GC.GetTotalMemory(false);

Console.WriteLine("{0} bytes per double", ((bytes2 - bytes1) / (1000 \* 1000)));

}

}

**Output**

1.5

-1

True

99

System.Double

System.Double

-1.79769313486232E+308

1.79769313486232E+308

8 bytes per double

**Parameters.**For developers concerned with performance, double has some drawbacks. When you pass a double as an argument, it is received as a formal parameter. This typically requires the bits to be copied into another memory location.

**And:** If you use an integer, only four bytes will be copied. If you use a double, eight bytes. The extra copying impacts performance.

**Parse, TryParse.**The double.Parse and double.TryParse methods are static—you call them on the "double" type. The difference between the two methods is that double.Parse throws exceptions, while double.TryParse does not.

[**Static Method**](https://www.dotnetperls.com/static)

**Next:** We see an example that demonstrates the different strings double.Parse methods handle.

[**Strings**](https://www.dotnetperls.com/string)

**C# program that uses double.Parse**

using System;

class Program

{

static void Main()

{

//

// Usage of double.Parse on various input strings.

//

string[] tests = new string[]

{

"1,000.00", // <-- This is 1000

"1.000", // <-- This is 1

"0.201", //

"00.001", // <-- This is 0.001

"-0.01", // <-- This is -0.01

"500000000", // <-- Five-hundred million

"0.0" // <-- 0

};

foreach (string test in tests)

{

double value = **double.Parse**(test);

Console.WriteLine(value);

}

//

// Usage of double.TryParse on various unusual inputs

//

string[] unusuals = new string[]

{

"NaN", // <-- This can be parsed.

"MaxValue", // <-- This fails.

"NegativeInfinity",

"Programmer",

"0.01-0.02",

" 0" // <-- This succeeds and is 0.

};

foreach (string unusual in unusuals)

{

double value;

if (**double.TryParse**(unusual, out value)) // Returns bool

{

Console.WriteLine("Valid: {0}", value);

}

}

}

}

**Output**

1000

1

0.201

0.001

-0.01

500000000

0

Valid: NaN

Valid: 0

**The double.TryParse method**is an instance of the tester-doer pattern. This pattern describes methods that see if some action can be done before actually doing it. This removes the possibility of a parsing error. [**Tester-Doer**](https://www.dotnetperls.com/tester-doer)

**Tip:** Using double.TryParse will enhance performance if you deal with lots of invalid input.

**Correctly parsed strings.**You can have numbers with commas for thousands, excess decimal places with zeros, excess leading zeros, negative signs, large numbers such as 500 million, zero with a decimal, and spaces surrounding the digits.

**Note:** The Convert.ToDouble method, when called with the string parameter overload, simply calls double.Parse internally after a null check.

**Therefore:** This method is not useful. But Convert.ToDouble also has other overloads that can be useful when not dealing with strings.

**These two parsing methods**are invaluable for when you are storing percentages in text files or databases. And for large numbers that are not monetary values, double.Parse is useful.

[**Percentage**](https://www.dotnetperls.com/percentage)

**However:** When dealing with currency values, we should investigate the decimal type and its parsing methods.

[**Decimal**](https://www.dotnetperls.com/decimal)

**Summary.**As a low-level type, the double uses eight bytes of memory. As a value type, it has clear efficiency advantages over any possible object-based replacements. For accurate representations, you can instead choose the decimal type.

**C# Example Pages**

**Percentages**are often useful in programs. With the numbers 1 and 2, we can get a percentage of 50%. We display and process percentages with doubles. We solve an annoying rounding problem.

[**Double**](https://www.dotnetperls.com/double)

**First,** we see some code that uses string.Format to display 2 numbers or ratio as a percentage. The following code shows 3 custom methods. These all handle percentages—but their logic is implemented in different ways.

**Note:** DisplayPercentage accepts a double that is a ratio. The {0:0.0%} indicates a percentage with 1 digit past the decimal place.

**Note 2:** The second DisplayPercentage accepts 2 parameters and then passes the ratio of them to the other method. It casts to double.

[**Casts**](https://www.dotnetperls.com/cast)

**Also:** GetPercentageString accepts a double containing a ratio and returns a percentage string using ToString().

[**ToString**](https://www.dotnetperls.com/tostring)

**Based on:** .NET (2019)

**C# program that calculates per cents**

using System;

class Program

{

static void Main()

{

// Display percentage of visits that resulted in purchases.

int purchases = 10;

int visitors = 120;

DisplayPercentage((double)purchases / visitors);

// Display 50 percent with overloaded method.

DisplayPercentage(1, 2);

// Write percentage string of nine tenths.

Console.WriteLine(GetPercentageString((double)9 / 10));

}

/// <summary>

/// This method writes the percentage form of a double to the console.

/// </summary>

static void **DisplayPercentage**(double ratio)

{

string percentage = string.Format("Percentage is {0:0.0%}", ratio);

Console.WriteLine(percentage);

}

/// <summary>

/// This method writes the percentage of the top number to the bottom number.

/// </summary>

static void **DisplayPercentage**(int top, int bottom)

{

DisplayPercentage((double)top / bottom);

}

/// <summary>

/// This method returns the percentage-formatted string.

/// </summary>

static string **GetPercentageString**(double ratio)

{

return ratio.ToString("0.0%");

}

}

**Output**

Percentage is 8.3%

Percentage is 50.0%

90.0%

**Example 2.**Here we convert two integers into a percentage manually with division and multiplication. Sometimes you can need raw percentages when you have percentages in the database stored in different formats.

**C# program that converts ratios**

using System;

class Program

{

static void Main()

{

// We want to have 92.9% from these two numbers.

int valid = 92;

int total = 99;

// First multiply top by 100 then divide.

double percent = (double)(valid \* 100) / total; // <-- Use cast

// This is the percent number.

Console.WriteLine(percent);

Console.WriteLine(Math.Floor(percent));

Console.WriteLine(Math.Ceiling(percent));

Console.WriteLine(Math.Round(percent, 1));

}

}

**Output**

92.9292929292929

92

93

92.9

**Casting to double.** The double must be assigned to a value cast to double. If you omit the cast, your value will be rounded and probably useless. When casting to double, you do not need to surround the entire expression with parentheses. [**Divide**](https://www.dotnetperls.com/divide)

**The final four statements**in the program display different forms of the percentage. Math.Floor rounds down to the nearest integer. Math.Ceiling rounds up to the nearest integer. And Math.Round rounds to a single decimal place.

[**Math.Floor**](https://www.dotnetperls.com/math-floor)[**Math.Ceiling**](https://www.dotnetperls.com/math-ceiling)[**Math.Round**](https://www.dotnetperls.com/math-round)

**Modulo.**The percentage sign in the C# language has a use as the modulo operator. This forms an expression that will return the remainder of a division of the two operands. You can find more about the modulo operator.

[**Modulo**](https://www.dotnetperls.com/modulo)

**Tip:** With modulo division, we can run an operation every N times. This has uses in many programs.

**Summary.**We saw two examples of using percentages in the C# language. First we saw how to format ratios as percentages with three different methods. Second, we saw how to get a percentage value directly with math, and then round it.

**Math.Floor**rounds down. It operates on types such as decimal or double. It reduces the value to the nearest integer. The Floor method is straightforward, but useful, when it is called for in programming.

[**Math.Ceiling**](https://www.dotnetperls.com/math-ceiling)

**Example.**First, the Floor method is found in the base class library and is available in the Math type. It implements the mathematical floor function, which finds the largest integer "not greater" than the original number.

**Next:** This example shows the Floor method being used on doubles that would be rounded down or rounded up with the Math.Round method.

[**Double**](https://www.dotnetperls.com/double)

**Based on:** .NET (2019)

**C# program that uses Math.Floor**

using System;

class Program

{

static void Main()

{

//

// Two values.

//

double value1 = 123.456;

double value2 = 123.987;

//

// Take floors of these values.

//

double floor1 = **Math.Floor**(value1);

double floor2 = **Math.Floor**(value2);

//

// Write first value and floor.

//

Console.WriteLine(value1);

Console.WriteLine(floor1);

//

// Write second value and floor.

//

Console.WriteLine(value2);

Console.WriteLine(floor2);

}

}

**Output**

123.456

123 [floor]

123.987

123 [floor]

**In the example,** the two numbers 123.456 and 123.987 are rounded down to the nearest integer. This means that regardless of how close they are close to 124, they are rounded to 123.

**Note:** Floor can be useful when rounding numbers that are part of a larger representation of another number.

**Discussion.**The Math.Floor method when given a positive number will erase the digits after the decimal place. But when it receives a negative number, it will erase the digits and increase the number's negativity by 1.

**So:** Using Math.Floor on a negative number will still decrease the total number. This means it will always become smaller.

**Next,**the Math.Floor method can be used on the 128-bit decimal type. When you call Math.Floor with this data type, the decimal.Floor static method is immediately called into. It is sometimes clearer to directly use decimal.Floor.

[**Decimal**](https://www.dotnetperls.com/decimal)[**Static Method**](https://www.dotnetperls.com/static)

**Summary.**We looked at the Math.Floor method in the base class library. This method is occasionally useful when you are trying to represent ratios and percentages and do not want the figures to add up to greater than 1 or 100%.

**Tip:** We can use Math.Floor on the double type or decimal type, which calls into the decimal.Floor method.

**Math.Ceiling** rounds up to the next full integer. This means that any number over 1 but under 2 would be rounded to 2. This is not the same result from rounding a number. Math.Ceiling is considered a ceiling function in mathematics.

**Example.**First, the Math.Ceiling method in the System namespace is a static method that returns a value type. The method receives a double or decimal type which is resolved at compile-time.

[**Static Method**](https://www.dotnetperls.com/static)

**Then:** It performs the computation and returns a new value as the result type at runtime.

**The method**copies the actual values and no allocations are done internally. This example shows that the number 123.456 has the ceiling of 124. We compute this with Math.Ceiling and display the result with Console.WriteLine.

[**Console.WriteLine**](https://www.dotnetperls.com/console)

**Based on:** .NET (2019)

**C# program that uses Ceiling**

using System;

class Program

{

static void Main()

{

// Get ceiling of double value.

double value1 = 123.456;

double ceiling1 = **Math.Ceiling**(value1);

// Get ceiling of decimal value.

decimal value2 = 456.789M;

decimal ceiling2 = **Math.Ceiling**(value2);

// Get ceiling of negative value.

double value3 = -100.5;

double ceiling3 = **Math.Ceiling**(value3);

// Write values.

Console.WriteLine(value1);

Console.WriteLine(ceiling1);

Console.WriteLine(value2);

Console.WriteLine(ceiling2);

Console.WriteLine(value3);

Console.WriteLine(ceiling3);

}

}

**Output**

123.456

124

456.789

457

-100.5

-100

**In this example,**different versions of Math.Ceiling are called because the compiler applies overload resolution each time. The ceiling of the number 123.456 is 124. The decimal type with value 456.789 has a ceiling of 457.

[**Double Type**](https://www.dotnetperls.com/double)[**Decimal Type**](https://www.dotnetperls.com/decimal)

**Negative ceiling values.** When you call the Math.Ceiling method on a negative floating point type, the number will be also be rounded up, which means that the ceiling of -100.5 is -100.

**Note:** For negative numbers, the digits after the decimal place are removed but the final result is not incremented.

**Implementation.** We review the internal implementation of the Math.Ceiling method in the base class library. The overload for Math.Ceiling that acts on double types will call into a hidden method that is not represented in managed code.

**So:** Math.Ceiling is likely to be far more optimized than any other method you could develop in C# code.

**For decimal,** the decimal.Ceiling method is invoked, which calls into the explicit unary negation operator on the decimal type. The ceiling is found by negating the value, taking its floor, and then negating the result again.

**Uses.**Here we discuss some usages of the Math.Ceiling and Math.Floor methods in the C# language. When you are reporting percentages based on data, using Math.Ceiling or Math.Floor is useful—they can help make the output more consistent.

[**Math.Floor**](https://www.dotnetperls.com/math-floor)

**Also,**when you are trying to display numbers that together represent another number, carefully controlling rounding with Ceiling and Floor is useful. This is because it avoids errors.

**Summary.**We looked at the functionality of the Math.Ceiling method. This function always rounds up a number, even when used on a negative number. We noted the implementation and some possible uses of the ceiling and floor functions in software.

**Math.Round.**This method rounds numbers to the nearest value. It receives the desired number of significant digits. It is part of the System namespace.

**Method details.** This Math.Round static method provides an accurate way to round double and decimal types. It reduces the risk of bugs.

[**Static Method**](https://www.dotnetperls.com/static)

**First example.**Math.Round has several overloads and 2 rounding modes defined on the MidpointRounding enum. Here we round an example double and decimal type.

**Return:** We see what Math.Round will return with different arguments. We use 1, 2 and 3 arguments.

[**Overload**](https://www.dotnetperls.com/overload)[**Double**](https://www.dotnetperls.com/double)[**Decimal**](https://www.dotnetperls.com/decimal)

**Based on:** .NET (2019)

**C# program that uses Math.Round**

using System;

class Program

{

static void Main()

{

//

// Round double type in three ways.

//

double before1 = 123.45;

double after1 = **Math.Round**(before1, 1,

MidpointRounding.AwayFromZero); // Rounds "up"

double after2 = **Math.Round**(before1, 1,

MidpointRounding.ToEven); // Rounds to even

double after3 = **Math.Round**(before1);

Console.WriteLine(before1); // Original

Console.WriteLine(after1);

Console.WriteLine(after2);

Console.WriteLine(after3);

//

// Round decimal type.

//

decimal before2 = 125.101M;

decimal after4 = **Math.Round**(before2);

decimal after5 = **Math.Round**(before2, 1);

Console.WriteLine(before2); // Original

Console.WriteLine(after4);

Console.WriteLine(after5);

}

}

**Output**

123.45

123.5 <-- MidpointRounding.AwayFromZero

123.4 <-- MidpointRounding.ToEven

123

125.101

125

125.1

**Notes, above program.** Math.Round is called several times. Because the .NET Framework differentiates between a decimal type and a double type, the best overloads are automatically chosen.

**Argument 1:** The first argument is the number you are rounding. This can be a double or decimal type.

**Argument 2:** The second argument is the number of digits after the decimal place you want to keep.

**Argument 3:** And the third argument is an optional rounding mode enumerated constant.

[**Enum**](https://www.dotnetperls.com/enum)

**Notes, continued.**The program shows 3 outputs of Math.Round on a double with value 123.45. It uses the MidpointRounding.AwayFromZero and MidpointRounding.ToEven constants.

**MidpointRounding.**What is the difference between MidpointRounding.ToEven and MidpointRounding.AwayFromZero?

My testing indicates that the difference is found when rounding the number 0.5.

**Info:** MidpointRounding.ToEven will round 0.5 to 0—this is because zero is even.

**And:** MidpointRounding.AwayFromZero will round 0.5 to 1—this is because one is away from zero.

**C# program that demonstrates MidpointRounding**

using System;

class Program

{

static void Main()

{

for (double i = 0.1; i < 0.99; i += 0.1)

{

Console.WriteLine("{0}=({1},{2})", i,

Math.Round(i, **MidpointRounding.ToEven**),

Math.Round(i, **MidpointRounding.AwayFromZero**));

}

}

}

**Output**

0.1=(0,0)

0.2=(0,0)

0.3=(0,0)

0.4=(0,0)

0.5=(0,1)

0.6=(1,1)

0.7=(1,1)

0.8=(1,1)

0.9=(1,1)

**Performance.** How does Math.Round affect performance? Recently I added calls to Math.Round in a program that is performance-sensitive.

**Further:** How did this affect the overall efficiency of the program? I benchmarked Math.Round to get a general idea.

**C# program that benchmarks Math.Round**

using System;

using System.Diagnostics;

class Program

{

const int \_max = 100000000;

static void Main()

{

var s1 = Stopwatch.StartNew();

for (int i = 0; i < \_max; i++)

{

double d = **Math.Round**(1.3665, 0);

if (d == 1.5)

{

throw new Exception();

}

}

s1.Stop();

var s2 = Stopwatch.StartNew();

for (int i = 0; i < \_max; i++)

{

double d = **Math.Round**(1.3665, 1);

if (d == 1.5)

{

throw new Exception();

}

}

s2.Stop();

Console.WriteLine(((double)(s1.Elapsed.TotalMilliseconds \* 1000000) /

\_max).ToString("0.00 ns"));

Console.WriteLine(((double)(s2.Elapsed.TotalMilliseconds \* 1000000) /

\_max).ToString("0.00 ns"));

Console.Read();

}

}

**Output**

20.72 ns Math.Round, 0

25.26 ns Math.Round, 1

**Notes, results.** Calls to Math.Round required about 20 nanoseconds. A call with zero decimal places required less time than a call with one decimal place.

**Tip:** It is possible that the more decimal places required, the slower the method becomes.

**Also:** Certain computations could be optimized by avoiding Math.Round. An if-statement would evaluate faster.

[**If**](https://www.dotnetperls.com/if)

**Cast:** Simply casting a double to an int rounds down. This requires only 2 nanoseconds. Using Math.Round requires 20 nanoseconds.

[**Cast, Int**](https://www.dotnetperls.com/cast-int)

**Note:** This article had an error in its description of rounding behaviors on scientific data. Thanks to Gus Gustafson for a correction.

**A summary.**We looked at the Math.Round method. We saw an example of the MidpointRounding enumerated type and how you can round numbers "away from zero" and to the nearest even number.

**Modulo.**This operator gets a remainder. It provides a way to execute code once every several iterations of a loop. It uses the percentage sign character in the lexical syntax.

**Modulo, notes.**Modulo has some unique properties. As with all low-level operations, it has a specific cost. We analyze modulo division in the C# language.

**Example.**Modulo division is expressed with the percentage sign. It is implemented with the rem instruction in the intermediate language. Rem takes the top 2 values on the evaluation stack.

**Then:** Rem performs the computation that returns the remainder of the division. It pushes that value onto the evaluation stack.

**Here:** This example demonstrates the math behind modulo. The expressions here are turned into constants during the C# compilation step.

**And:** No rem instructions are generated. Programs are evaluated in many phases, but the end result should always make sense.

**Based on:** .NET (2019)

**C# program that uses modulo operator**

using System;

class Program

{

static void Main()

{

//

// When 1000 is divided by 90, the remainder is 10.

//

Console.WriteLine(1000 % 90);

//

// When 100 is divided by 90, the remainder is also 10.

//

Console.WriteLine(100 % 90);

//

// When 81 is divided by 80, the remainder is 1.

//

Console.WriteLine(81 % 80);

//

// When 1 is divided by 1, the remainder is zero.

//

Console.WriteLine(1 % 1);

}

}

**Output**

10

10

1

0

**Notes, above program.** The program shows the remainders of the divisions of the 2 integers at each step. The runtime never performs modulo divisions here. The C# compiler does.

**Important:** We see that 1000 and 100 divide into parts of 90 with a remainder of 10.

**Note:** If the first argument to the predefined modulo operator is 81 and the second operand is 80, the expression evaluates to a value of 1.

**Notes, continued.**If you use modulo on the same 2 operands, you receive 0 because there is no remainder. If you use modulo by 0, you will get a compile error or a runtime exception.

[**DivideByZeroException**](https://www.dotnetperls.com/dividebyzeroexception)[**Compile-Time Error**](https://www.dotnetperls.com/compile-time-error)

**Example 2.**You can apply modulo in a loop to achieve an interval or step effect. If you use a modulo operation on the loop index variable, you can execute code at an interval.

**Note:** This example shows how to write to the screen every ten iterations in the for-loop.

**C# program that uses modulo division in loop**

using System;

class Program

{

static void Main()

{

//

// Prints every tenth number from 0 to 200.

// Includes the first iteration.

//

**for** (int i = 0; i < 200; i++)

{

if ((i % 10) == 0)

{

Console.WriteLine(i);

}

}

}

}

**Output**

0

10

20

30

40

50

60

70

80

90

100

110

120

130

140

150

160

170

180

190

**Notes, if.**Often, modulo divisions are performed in ifs and used in control flow. The three numbers in the condition in the if-statement can have any values, but we cannot divide by 0.

**Discussion.**Modulo has several common uses in programs. You can use modulo division in loops to only execute code every several iterations. This can improve real code.

**Note:** We do not often need to compute numeric remainders for user consumption. The regular division operator may be more useful here.

[**Divide**](https://www.dotnetperls.com/divide)

**Odd:** You can use modulo to test for odd numbers and even numbers. You can define odd numbers as not-even numbers.

[**Odd, Even**](https://www.dotnetperls.com/odd)

**Performance.** Modulo is slower than other arithmetic operators such as increment and decrement or even multiply. This is a hardware limitation on computers.

**But:** The total time required for individual modulo operations is tiny compared to other tasks such as disk reads or network accesses.

**So:** If you can reduce those operations with modulo division, you can improve overall performance.

**Performance, table.**The time required for modulo division depends on hardware and other factors. Some analysis from Microsoft is helpful. This page provides a table listing times required.

[**Writing Faster Managed Code: Microsoft Docs**](https://docs.microsoft.com/en-us/previous-versions/dotnet/articles/ms973852(v=msdn.10))

**Estimated costs of instructions:**

Add: 1 ns

Subtract: 1 ns

Multiply: 2.7 ns

Divide: 35.9 ns

**Performance, loops.**You may rarely have a modulo division in a hot path and this can cause a loss of performance. This will almost always occur in a loop body or in a recursive method.

**Tip:** You can apply a technique called "strength reduction" manually to convert the modulo operation into a subtraction or addition.

**And:** To do this, add another field or local variable. Then, in each iteration of the loop, decrement it and test it against zero.

**Then:** When zero is reached, set it to its maximum value again. This resets the pattern.

**A summary.**The modulo operator is often useful. This is implemented in the CLI as a rem instruction. The C# compiler calculates modulo divisions of constants at compile-time.

**Modulo division** returns the remainder of the 2 operands. We use the "percent" symbol for modulo in the C# language. This is a powerful operator, but it has its nuances.

**Casts.** Casting variables is complex. A set of rules resolves casts. In some cases data is lost and the cast cannot be reversed. In others an exception is provoked.

**Conversions,** specified as methods or operators, are often required. Lists and arrays are similar, but not the same. To convert, we use ToList and ToArray.

**A first program.**We cast a double value to an int. This is an explicit numeric cast. Some casts may be implicit (not specified in the syntax).

**Is, as:** With the is-operator we cast an object reference. Finally, we cast that same reference with the as-operator.

**So:** The StringBuilder is converted to an object and then back into a StringBuilder reference.

**Based on:** .NET (2019)

**C# program that uses casts**

using System;

using System.Text;

class Program

{

static void Main()

{

int value = (int)1.5; // Cast 1.

Console.WriteLine(value);

object val = new StringBuilder();

if (val **is** StringBuilder) // Cast 2.

{

StringBuilder builder = val **as** StringBuilder; // Cast 3.

Console.WriteLine(builder.Length == 0);

}

}

}

**Output**

1

True

**Implicit, explicit.**Implicit casts are not visible in the source text. The explicit (int) on the left side of a variable or expression casts to an int.

**Example:** Here the integer cast succeeds. But we then try to cast to an invalid type.

**Warning:** If you incorrectly use a cast expression, you will cause an InvalidCastException to be thrown.

[**InvalidCastException**](https://www.dotnetperls.com/invalidcastexception)

**Implicit casts:** Intended never to provoke an exception. These are used when a value is expanded to more bytes.

**Explicit casts:** Allowed to provoke an exception. Explicit casts are used when a value is reduced to fewer bytes.

**C# program that uses cast expressions**

using System;

class Program

{

static void Main()

{

// Assign an int and then cast it to an object implicitly.

int value1 = 400;

object value2 = **value1**;

// Explicitly cast to an integer again.

int value3 = (int)value2;

Console.WriteLine(value3);

// Try to cast it to a string.

try

{

string value4 = (string)value2;

}

catch (Exception ex)

{

Console.WriteLine(ex);

}

}

}

**Output**

400

System.InvalidCastException:

Unable to cast object of type 'System.Int32' to type 'System.String'.

at Program.Main()

**As-operator.** This cast allow us to store the result of the cast in a local variable, in a single expression. I recommend "as" for all reference types.

[**as**](https://www.dotnetperls.com/as)

**Also:** The as-cast will result in exception-neutral code—which is also faster code.

**Is-operator.** This is the same as the as-operator but it returns true or false, not a variable reference. If we use "is," we may end up casting too many times.

[**is**](https://www.dotnetperls.com/is)

**Numeric casts.**Many casts apply to number types. We cast numeric values. We explore the intermediate language's casting instructions.

[**Numeric Casts**](https://www.dotnetperls.com/numeric-casts)

**Cast to int.** We can cast fractional values, like doubles, to ints. This is a useful technique for rounding numbers. It can be used as an optimization.

[**Cast, Int**](https://www.dotnetperls.com/cast-int)

**Type hierarchy.** This is used to specify behavior based on the structure of a program. We specify differences in methods called through the derivation chain.

**Note:** Type hierarchies are implemented with a matrix. In casts, this matrix determines if the cast succeeds. This is "transitive closure."

**Convert.**A List is not an array. This is no problem unless we need an array. Conversions are possible for both simple values and complex data types.

**Examples:** These examples show how to convert types. The provided conversions are more complex than casts.

**Here:** In this program, we use an extension from the System.Linq namespace to convert an array into a List.

[**LINQ**](https://www.dotnetperls.com/linq)

**C# program that converts array to List**

using System;

using System.Collections.Generic;

using System.Linq;

class Program

{

static void Main()

{

int[] array = { 1, 2, 3 };

List<int> list = array.**ToList**();

Console.WriteLine(array[0] == list[0]);

Console.WriteLine(array.Length == list.Count);

}

}

**Output**

True

True

**List.**This is a generic collection. Often we must convert Lists to Dictionaries, strings, and DataTables. And the opposite conversions too are needed.

[**List to Dictionary**](https://www.dotnetperls.com/convert-dictionary-list)[**List to DataTable**](https://www.dotnetperls.com/convert-list-datatable)[**List to String**](https://www.dotnetperls.com/convert-list-string)

**Arrays.**Some of the more complicated conversion methods involve an array type. We convert various types, such as ArrayList, string, and List, into arrays and back again.

[**ArrayList to Array**](https://www.dotnetperls.com/convert-arraylist-array)

[**ArrayList to List**](https://www.dotnetperls.com/convert-arraylist-list)

[**Char Array to String**](https://www.dotnetperls.com/convert-char-array-string)

[**List to Array**](https://www.dotnetperls.com/convert-list-array)[**String Array to String**](https://www.dotnetperls.com/convert-string-array-string)

[**String to Byte Array**](https://www.dotnetperls.com/convert-string-byte-array)

**BitConverter.**This class converts byte arrays into integral value types, and the opposite. This tutorial uses and tests BitConverter.

[**BitConverter**](https://www.dotnetperls.com/bitconverter)

**Methods versus casts.** For many classes, custom conversion methods, even complex ones, are needed. But for simpler value types, using a casting expression is sufficient. It is also faster.

**Numeric.**Most numeric conversions involve a multiplication or division expression. I developed (and tested) some reusable unit conversion methods.

[**Bytes, Megabytes, Gigabytes**](https://www.dotnetperls.com/convert-bytes-megabytes)

[**Celsius to Fahrenheit**](https://www.dotnetperls.com/convert-degrees)

[**Days to Months**](https://www.dotnetperls.com/convert-days-months)

[**Feet to Inches**](https://www.dotnetperls.com/feet-inches)

[**Miles to Kilometers**](https://www.dotnetperls.com/convert-miles-kilometers)

[**Milliseconds to Seconds**](https://www.dotnetperls.com/convert-milliseconds)

[**Nanoseconds to Milliseconds**](https://www.dotnetperls.com/convert-nanoseconds)

**Bools.** We cannot directly convert bools and ints. The .NET Framework disallows this. Instead we must use expressions to convert them as needed.

[**Bool to Int**](https://www.dotnetperls.com/convert-bool-int)[**Int to Bool**](https://www.dotnetperls.com/bool)

**TimeSpan.** Our programs use many built-in structs like TimeSpan. We can convert TimeSpan into a long value. This may be easier to persist to disk.

[**TimeSpan to long**](https://www.dotnetperls.com/convert-timespan-long)

**Strings.**Often our programs have string data that represents numbers or times. With int.Parse and TryParse, we can parse that data into an int.

[**String to int**](https://www.dotnetperls.com/parse)

**Strings, arrays, collections.**With ToCharArray we convert a string into its equivalent char array. Custom methods, which often use Split, can parse more complex formats.

[**Char to String**](https://www.dotnetperls.com/convert-char-string)[**Dictionary to String**](https://www.dotnetperls.com/convert-dictionary-string)

**ToCharArray:** This method is provided on the string class. It returns the underlying buffer of a string as a char array.

[**ToCharArray**](https://www.dotnetperls.com/tochararray)

**C# program that converts string, array**

using System;

class Program

{

static void Main()

{

// A string value.

**string** value = "test";

// Convert the string into an array.

char[] array = value.**ToCharArray**();

// Display parts of the array.

Console.WriteLine(array[0]);

Console.WriteLine(array.Length);

}

}

**Output**

t

4

**Char.**All chars have an underlying integer representation. A lowercase a, for example, is 97. We can implicitly cast a char to an int, as an int is larger.

[**Char**](https://www.dotnetperls.com/char)

**However:** To cast from an int to a char, a cast is needed. This is a narrowing conversion where data loss is possible.

**Sizes:** A char is only 2 bytes. But an int is 4 bytes—so larger numbers in an int cannot be safely turned into chars.

**C# program that casts char**

using System;

class Program

{

static void Main()

{

for (char c = 'a'; c <= 'e'; c++)

{

// Cast the char to an int.

int code = c;

// Cast the int to a char.

// ... An int is larger than a char.

// ... So an explicit cast is needed.

char original = (char)c;

Console.WriteLine(c + "..." + code + "..." + original);

}

}

}

**Output**

a...97...a

b...98...b

c...99...c

d...100...d

e...101...e

**Operators.**We can define conversion operators with method bodies. Implicit conversions require no special syntax. Explicit ones require a cast expression.

[**Explicit, Implicit**](https://www.dotnetperls.com/explicit)

**In compiler textbooks,**we learn that an implicit conversion is one done automatically. This kind of cast is called a "coercion." The C# compiler limits coercions to lossless operations.

**Quote:** Conversion from one type to another is said to be implicit if it is done automatically by the compiler. Implicit type conversions, also called coercions, are limited in many languages to widening conversions (Compilers: Principles, Techniques and Tools).

**Cast expressions abound.** We demonstrated explicit and implicit casts. An explicit cast expression can provoke a runtime exception. An implicit one does not.

**To sum up:**casting relies on the type hierarchy. With the "is" and "as" operators we safely cast. For complex conversions, custom methods are needed.