## **Working with Numbers**

Understanding how to effectively work with numbers is fundamental to almost any software engineering task. C# and the .NET platform provide a robust set of features, from simple conversions to handling arbitrarily large numbers and complex mathematical constructs.

### **Numeric Conversions: A Quick Recap**

* **Parsing Base 10 Numbers:**
  + Parse(): Converts a string to a numeric type. It will throw a FormatException if the string cannot be parsed.
  + TryParse(): A safer alternative that returns a boolean indicating success or failure and provides the result via an out parameter. This is generally preferred for robust error handling.
  + **Example:** double d = double.Parse("3.5"); or int i; bool ok = int.TryParse("3", out i);
* **Parsing from Other Bases (Binary, Octal, Hexadecimal):**
  + Convert.ToInt32(string s, int base) (and similar Convert.To... methods for other integral types): This method allows you to parse a string representing a number in a specified base (2, 8, 10, or 16).
  + **Example:** int i = Convert.ToInt32("1E", 16); (This will parse the hexadecimal string "1E" as the decimal integer 30).
* **Formatting to Hexadecimal:**
  + ToString("X") (for uppercase hexadecimal) or ToString("x") (for lowercase hexadecimal) on an integral type.
  + **Example:** string hex = 45.ToString("X"); (This will result in the string "2D").
* **Lossless Numeric Conversion:**
  + **Implicit Cast:** These conversions occur automatically when converting from a smaller numeric type to a larger one where no loss of precision or range is possible (e.g., int to double).
  + **Example:** int i = 23; double d = i;
* **Truncating Numeric Conversion:**
  + **Explicit Cast:** Required when converting from a larger numeric type to a smaller one where data might be lost (e.g., double to int). This operation simply discards any fractional part.
  + **Example:** double d = 23.5; int i = (int)d; (The value of i will be 23).
* **Rounding Numeric Conversion (Real to Integral):**
  + Convert.ToInt32(double value) (and similar Convert.To... methods): These methods convert a floating-point number to an integer by rounding. Crucially, they employ **banker's rounding** for midpoint values (e.g., 2.5, 3.5), rounding to the nearest *even* integer to avoid bias.
  + **Example:** double d = 23.5; int i = Convert.ToInt32(d); (The value of i will be 24, as 24 is the nearest even number).

### **The Math Class: Standard Mathematical Functions**

The System.Math class is a static utility class that provides a comprehensive set of commonly used mathematical functions and constants.

* **Key Categories and Methods:**
  + **Rounding:** Round(), Truncate(), Floor(), Ceiling().
    - Round() offers control over decimal places and the rounding convention for midpoints.
    - Floor() always rounds down to the nearest integer.
    - Ceiling() always rounds up to the nearest integer.
  + **Maximum/Minimum:** Max(), Min(). Note that these methods accept only two arguments. For finding the maximum or minimum in an array or collection, leverage the Max() and Min() extension methods from System.Linq.Enumerable.
  + **Absolute Value and Sign:** Abs(), Sign().
  + **Square Root:** Sqrt().
  + **Raising to a Power:** Pow(), Exp().
  + **Logarithm:** Log(), Log10().
  + **Trigonometric Functions:** Sin(), Cos(), Tan(), Sinh(), Cosh(), Tanh(), Asin(), Acos(), Atan(). Most of these accept double arguments.
  + **Mathematical Constants:** Math.E (Euler's number) and Math.PI.

### **BigInteger: Handling Arbitrarily Large Integers**

The System.Numerics.BigInteger struct is a specialized numeric type designed to represent integers of any arbitrary size without loss of precision, far exceeding the limits of long. This is indispensable for tasks like cryptography, very large-scale calculations, or scenarios where integer overflow is unacceptable.

* **No Native Literals:** C# does not provide direct literal support for BigInteger.
* **Construction:**
  + Implicit conversion from any standard integral type.

| BigInteger twentyFive = 25; |
| --- |

* + Static methods like BigInteger.Pow(base, exponent) for large powers.

| BigInteger googol = BigInteger.Pow(10, 100); // Represents 10^100 |
| --- |

* + Parsing from a string using BigInteger.Parse(string s).

| BigInteger googol = BigInteger.Parse("1".PadRight(101, '0')); // A googol as a string |
| --- |

* **Output:** ToString() will print every single digit of the number.
* **Conversions:** Explicit cast operators are used for potentially lossy conversions between BigInteger and standard numeric types.

| double g2 = (double)googol; // Will incur precision loss for such a large number |
| --- |

* **Operators:** BigInteger overloads all standard arithmetic, comparison, and equality operators, allowing you to perform calculations naturally.
* **Byte Array Conversion:** You can construct a BigInteger from a byte array and convert it back using ToByteArray(), which is particularly useful for cryptographic applications.

### **Half: The 16-bit Floating-Point Type (Introduced in .NET 5)**

The System.Half struct represents a 16-bit floating-point number. Its primary utility lies in interoperability with hardware, such as graphics card processors, which often utilize half-precision floats for efficiency.

* **Conversion:** You can convert between Half and float or double using explicit casts.

| Half h = (Half)123.456f; Console.WriteLine(h); // Note: May show loss of precision, e.g., 123.44 |
| --- |

* **No Arithmetic Operations:** Half does not define its own arithmetic operators. To perform calculations, you must cast a Half value to float or double, perform the operation, and then cast the result back to Half if desired, accepting the inherent precision limitations.
* **Range:** It has a limited range, approximately from -65500 to 65500.

### **Complex: Representing Complex Numbers**

The System.Numerics.Complex struct is another specialized numeric type designed to represent **complex numbers**, which consist of both a real and an imaginary component (each represented as a double). This is crucial for applications in fields like signal processing, quantum mechanics, and advanced mathematics.

* **Construction:** Instantiate Complex by providing the real and imaginary values. Implicit conversions from standard numeric types are also available.

| var c1 = new Complex(2, 3.5); // Represents 2 + 3.5i var c2 = new Complex(3, 0); // Represents 3 + 0i (a purely real number) |
| --- |

* **Properties:** Access the Real and Imaginary components, as well as the Phase (angle) and Magnitude (length from the origin).
* **FromPolarCoordinates:** Construct a complex number by specifying its magnitude and phase.
* **Overloaded Operators:** All standard arithmetic operators (+, -, \*, /) are overloaded to work correctly with Complex numbers.

| Console.WriteLine(c1 + c2); // (5, 3.5) Console.WriteLine(c1 \* c2); // (6, 10.5) |
| --- |

* **Static Methods:** The Complex struct also offers static methods for more advanced functions, including trigonometric, logarithmic, and exponentiation operations, as well as the Conjugate.

### **Random: Pseudorandom Number Generation**

The System.Random class is used to generate sequences of pseudorandom numbers. It's suitable for most general-purpose applications like simulations, games, or generating test data.

* **Instantiation:**
  + new Random(): Initializes the random number generator using the current system time as a seed.
  + new Random(int seed): Allows you to provide a specific integer seed. Using the same seed will produce the exact same sequence of "random" numbers, which is useful for reproducibility in testing.
* **Crucial Best Practice:** Due to the limited granularity of the system clock, creating multiple Random instances in rapid succession (e.g., within a loop) without a specified seed will likely result in them producing identical sequences. **The recommended pattern is to declare a single, static Random instance and reuse it throughout your application.** In multithreaded environments, special care (like thread-local storage) is required as Random is not thread-safe.
* **Methods:**
  + Next(n): Generates a random integer between 0 (inclusive) and n (exclusive).
  + NextDouble(): Generates a random double between 0.0 (inclusive) and 1.0 (exclusive).
  + NextBytes(byte[] buffer): Fills a provided byte array with random values.
* **New in .NET 8:**
  + GetItems(collection, count): Selects n random items from a collection.
  + Shuffle(arrayOrSpan): Randomizes the order of elements within an array or span.
* **Cryptographic Randomness:** For high-security applications (e.g., cryptography), Random is not sufficient. Use System.Security.Cryptography.RandomNumberGenerator, which provides cryptographically strong random numbers, typically by filling a byte array. You would then convert these bytes to other numeric types using BitConverter.

### **BitOperations: Efficient Bitwise Operations (Introduced in .NET 6)**

The System.Numerics.BitOperations class provides highly optimized methods for various bitwise operations, often leveraging CPU intrinsics for maximum performance. These are valuable in low-level programming, data manipulation, and algorithms where direct bit manipulation is required.

* IsPow2(): Determines if a number is a power of 2.
* LeadingZeroCount()/TrailingZeroCount(): Returns the count of leading or trailing zero bits in an unsigned integer.
* Log2(): Returns the integer base-2 logarithm of an unsigned integer.
* PopCount(): Counts the number of bits set to 1 (also known as Hamming weight).
* RotateLeft()/RotateRight(): Performs bitwise left/right rotation.
* RoundUpToPowerOf2(): Rounds an unsigned integer up to the closest power of 2.