Foundations of C#

Variables and Data Types

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Variables and Their Significance

A variable in programming is a symbolic name associated with a storage location in memory that holds a data value. This value can be modified during the execution of a program. Variables are fundamental to programming because they allow developers to store, retrieve, and manipulate data dynamically. They are essential components in programming that serve several crucial functions:

Functions of Variables

- Variables hold data that can be used and modified throughout a program. The data type of a variable (e.g., integer, float, string) determines what kind of value it can store.
- When a variable is declared, the program allocates a specific portion of memory to store its value. This memory can be accessed and manipulated using the variable's name.
- Variables enable programs to handle data that can change during execution, such as user inputs, sensor readings, or the results of calculations.

Example of Variable Declaration and Initialization

```
int age = 25;
string name = "Alice";
bool isStudent = true;
```

- `int age = 25;` declares an integer variable named `age` and initializes it with the value 25.
- `string name = "Alice";` declares a string variable named `name` and initializes it with the value "Alice".
- `bool isStudent = true;` declares a boolean variable named `isStudent` and initializes it with the value `true`.

Variables provide a way to label data with descriptive names, making code more readable and maintainable.

Naming Conventions

Naming conventions are guidelines that help developers choose meaningful and consistent names for variables. In C#, variable names should be descriptive and follow specific rules to ensure clarity and avoid errors.

Rules for Naming Variables in C#

- C# is case-sensitive, so `variable` and `Variable` are considered different identifiers.
- Variable names can include letters, digits, and underscores (_), but cannot start with a digit.
- Variable names cannot contain spaces or special characters (e.g., @, #, $).
- Variable names cannot be C# reserved keywords (e.g., `int`, `class`, `public`).

Examples of Valid and Invalid Variable Names

Valid: `totalAmount`, `first_name`, `_counter`, `number1`
Invalid: `1stPlace`, `total-amount`, `first name`, `class`

Best Practices for Choosing Meaningful Variable Names

- Choose names that clearly describe the purpose of the variable.
  - **Good:** `totalAmount`, `userName`, `isStudent`
  - **Bad:** `x`, `tmp`, `a1`
- In C#, the common convention is to use camelCase for variable names, where the first letter is lowercase and each subsequent word starts with an uppercase letter.
  - **Example:** `numberOfItems`, `userAge`, `isLoggedIn`
- Avoid using abbreviations unless they are widely understood and accepted.
  - **Good:** `totalPrice`, `maximumValue`
  - **Bad:** `tp`, `maxVal`
- Be consistent with naming conventions throughout your codebase to improve readability and maintainability.
- Except for loop counters or temporary variables in short blocks of code, avoid single character names.
  - **Good:** `index`, `counter`
  - **Bad:** `i`, `n`
- Ensure that the variable name makes sense in the context in which it is used.
  - **Good:** `emailAddress`, `employeeCount`
  - **Bad:** `data`, `info`

Examples of Meaningful Variable Names

- `int numberOfStudents = 30;`
- `double averageTemperature = 23.5;`
- `string customerName = "John Doe";`
- `bool isEmailVerified = false;`

Variables are fundamental to programming in C#, providing a means to store and manipulate data dynamically. By adhering to proper naming conventions and choosing meaningful variable names, developers can write clear, readable, and maintainable code.

Declaring and Initializing Variables

Variable Declaration

Syntax for Declaring Variables in C#

In C#, declaring a variable involves specifying the data type of the variable followed by the variable's name. The declaration tells the compiler what type of data the variable will hold. The syntax for declaring a variable is:

```
dataType variableName;
```

Where `dataType` is the type of data the variable will store (e.g., int, double, string), and `variableName` is the identifier used to reference the variable.

Examples of Variable Declarations

- **Integer Declaration:**

  ```
  int age;
  ```

  This line declares a variable named `age` that will store integer values.

- **String Declaration:**

  ```
  string name;
  ```

  This line declares a variable named `name` that will store string values.

- **Boolean Declaration:**

  ```
  bool isStudent;
  ```

  This line declares a variable named `isStudent` that will store boolean values (true or false).

- **Double Declaration:**

  ```
  double temperature;
  ```

  This line declares a variable named `temperature` that will store double (floating-point) values.

Variable Initialization

Assigning Initial Values to Variables

Variable initialization involves assigning an initial value to a variable at the time of declaration. This is done using the assignment operator (=). The syntax for initializing a variable is:

```
dataType variableName = initialValue;
```

Where `initialValue` is the value assigned to the variable when it is declared.

Examples of Variable Initialization

- **Integer Initialization:**

  ```
  int age = 25;
  ```

  This line declares and initializes a variable named `age` with the value 25.

- **String Initialization:**

  ```
  string name = "Alice";
  ```

  This line declares and initializes a variable named `name` with the value "Alice".

- **Boolean Initialization:**

  ```
  bool isStudent = true;
  ```

  This line declares and initializes a variable named `isStudent` with the value `true`.

- **Double Initialization:**

  ```
  double temperature = 36.5;
  ```

  This line declares and initializes a variable named `temperature` with the value 36.5.

Differences Between Declaring and Initializing

Declaration Only

When a variable is declared but not initialized, it is allocated space in memory, but its value is not set.

- **Example:**

  ```
  int age;
  ```

  Here, `age` is declared but not assigned a value, so it holds the default value for its type (0 for integers).

Declaration with Initialization

When a variable is both declared and initialized, it is allocated space in memory and assigned an initial value.

- **Example:**

```
int age = 25;
```

Here, `age` is declared and initialized with the value 25.

Separate Declaration and Initialization

Variables can be declared first and then initialized later in the code.

- **Example:**

```
int age;
age = 25;
```

In this example, `age` is first declared and then initialized with the value 25 later in the code.

Practical Examples and Scenarios

Uninitialized Variable Use

```
int count;
// count is declared but not initialized
count = 10;
// count is now initialized with the value 10
```

Default Values

```
int defaultInt;
bool defaultBool;
double defaultDouble;
string defaultString;

// Default values
Console.WriteLine(defaultInt); // Output: 0
Console.WriteLine(defaultBool); // Output: False
Console.WriteLine(defaultDouble); // Output: 0
Console.WriteLine(defaultString); // Output: (null)
```

Multiple Declarations

```
int x, y, z;
// x, y, and z are all declared as integers
x = 1;
y = 2;
z = 3;
```

Inline Initialization

```
int x = 1, y = 2, z = 3;
// x, y, and z are declared and initialized inline
```

Properly declaring variables ensures that they are correctly typed and allocated in memory, while initialization assigns them meaningful values to be used in computations and logic.

Overview of Basic Data Types

Understanding the basic data types in C# is important for writing efficient and effective code. Each data type serves a specific purpose and has its own range of values and characteristics.

Signed and Unsigned Integer Data Types

In C#, integer data types can be classified as either signed or unsigned. The main difference between these two types lies in their ability to represent negative numbers.

Signed Integer Types

Signed integer types can represent both positive and negative values. They use the most significant bit (MSB) to indicate the sign of the number: 0 for positive and 1 for negative. The remaining bits are used to represent the magnitude of the number.

Signed Integer Types in C#:

- **sbyte (signed byte)**
  - Size: 8 bits (1 byte)
  - Range: -128 to 127
  - Example:

```
sbyte temperature = -10;
sbyte age = 25;
```

- **short (signed short)**
  - Size: 16 bits (2 bytes)
  - Range: -32,768 to 32,767
  - Example:

```
short elevation = -500;
short population = 1500;
```

- **int (signed integer)**
  - Size: 32 bits (4 bytes)
  - Range: -2,147,483,648 to 2,147,483,647

- Example:

```
int balance = -1000;
int score = 5000;
```

- **long (signed long)**
  - Size: 64 bits (8 bytes)
  - Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
  - Example:

```
long distance = -5000000000L;
long population = 7000000000L;
```

Unsigned Integer Types

Unsigned integer types can only represent non-negative values (positive values and zero). All bits are used to represent the magnitude of the number, allowing for a larger range of positive values compared to their signed counterparts.

Unsigned Integer Types in C#:

- **byte (unsigned byte)**
  - Size: 8 bits (1 byte)
  - Range: 0 to 255
  - Example:

```
byte level = 200;
byte age = 25;
```

- **ushort (unsigned short)**
  - Size: 16 bits (2 bytes)
  - Range: 0 to 65,535
  - Example:

```
ushort height = 500;
ushort width = 1500;
```

- **uint (unsigned integer)**
  - Size: 32 bits (4 bytes)
  - Range: 0 to 4,294,967,295
  - Example:

```
uint fileSize = 4000000000U;
uint count = 1000U;
```

- **ulong (unsigned long)**

- Size: 64 bits (8 bytes)
- Range: 0 to 18,446,744,073,709,551,615
- Example:

```
ulong distance = 5000000000UL;
ulong totalBytes = 18000000000000000000UL;
```

Differences and Use Cases

Range:

- Signed integers can represent both negative and positive numbers, with half the range dedicated to negative values and the other half to positive values.
- Unsigned integers can only represent non-negative numbers, effectively doubling the maximum positive value they can represent compared to their signed counterparts of the same size.

Use Cases:

- Signed integers are useful when negative values are expected or possible, such as in temperature readings, financial calculations (debits and credits), or any context where values can decrease below zero.
- Unsigned integers are ideal when only non-negative values are needed, such as for counting items, indexing arrays, or representing sizes, lengths, and other inherently non-negative quantities.

Examples Illustrating Differences:

- **Signed int:**

```
int balance = -100; // valid
int points = 150; // valid
```

- **Unsigned int:**

```
uint balance = -100; // invalid, will cause a compilation error
uint points = 150; // valid
```

Memory and Performance Considerations:

- Both signed and unsigned integers of the same size use the same amount of memory (e.g., int and uint both use 4 bytes).
- There is no significant performance difference between signed and unsigned integers in most cases. The choice between them should be based on the logical requirements of the application.

Precision with Floating-Point and Decimal Data Types

Floating-point and decimal data types in C# are used to represent numbers that have fractional components. These types include float, double, and decimal.

What is Precision?

Precision in floating-point and decimal numbers refers to the number of digits that can be accurately represented. This includes digits both before and after the decimal point. Precision determines how exact a number is when stored and manipulated in a program.

Floating-Point and Decimal Data Types

- **float (Single-Precision Floating-Point):**
  - Size: 32 bits (4 bytes)
  - Approximate Range: $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
  - Precision: Approximately 7 significant digits
- **double (Double-Precision Floating-Point):**
  - Size: 64 bits (8 bytes)
  - Approximate Range: $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$
  - Precision: Approximately 15-16 significant digits
- **decimal (Decimal Type):**
  - Size: 128 bits (16 bytes)
  - Approximate Range: $\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$
  - Precision: Approximately 28-29 significant digits

Understanding Precision and Accuracy

Precision: Indicates the number of significant digits with which a number is represented. For example, in a float, you might represent a number like 3.141592 accurately up to about 7 digits.

Accuracy: Refers to how close a number is to the true value. Due to precision limitations, floating-point numbers can only approximate many real numbers.

Examples Illustrating Precision

- **Using float:**

```
float num1 = 1.1234567f; // 7 digits of precision
float num2 = 1.12345678f; // 8 digits - less accurate due to
   precision limit
Console.WriteLine(num1); // Output: 1.123457 (approximated)
Console.WriteLine(num2); // Output: 1.123457 (approximated, last
   digit rounded)
```

- **Using double:**

```
double num1 = 1.123456789012345; // 15 digits of precision
double num2 = 1.1234567890123456; // 16 digits - less accurate due
    to precision limit
Console.WriteLine(num1); // Output: 1.123456789012345
Console.WriteLine(num2); // Output: 1.123456789012346 (approximated
    , last digit rounded)
```

- **Using decimal:**

```
decimal num1 = 1.12345678901234567890123456M; // 28 digits of
    precision
decimal num2 = 1.123456789012345678901234567M; // 29 digits - less
    accurate due to precision limit
Console.WriteLine(num1); // Output: 1.12345678901234567890123456
Console.WriteLine(num2); // Output: 1.12345678901234567890123457 (
    approximated, last digit rounded)
```

Common Issues Due to Precision Limitations

- **Rounding Errors:** Floating-point arithmetic can introduce rounding errors because not all decimal numbers can be represented precisely in binary form. For example:

```
double result = 0.1 + 0.2;
Console.WriteLine(result); // Output: 0.30000000000000004
```

- **Accumulation of Errors:** In iterative calculations or when combining many floating-point numbers, small errors can accumulate, leading to significant inaccuracies.

- **Comparison Problems:** Comparing floating-point numbers directly can be unreliable due to precision issues. Instead, a tolerance (epsilon) is used:

```
double a = 0.1 + 0.2;
double b = 0.3;
double epsilon = 0.0000001;
if (Math.Abs(a - b) < epsilon)
{
    Console.WriteLine("a and b are considered equal");
}
else
{
    Console.WriteLine("a and b are not equal");
}
```

Choosing Between float, double, and decimal

- **Use float when:**
  - Memory usage is a concern, and precision up to 7 digits is sufficient.
  - Applications like graphics or games where large arrays of floating-point numbers are used.

- **Use double when:**
  - Higher precision (up to 15-16 digits) is required.
  - Applications demand accurate numerical computations, such as scientific calculations, financial calculations, or simulations.
- **Use decimal when:**
  - Maximum precision is required (28-29 significant digits).
  - Applications involve financial calculations, currency computations, or other use cases requiring exact decimal representation to avoid rounding errors.

Precision in floating-point and decimal data types is a critical consideration when performing numerical calculations in C#. Understanding the limitations and appropriate uses of float, double, and decimal, and appropriately managing precision and accuracy, helps ensure reliable and accurate program behavior.

Character and String Data Types

Understanding the char and string data types is essential for handling text and character data in C#. These types are fundamental for any program that involves text processing, user input, or displaying messages.

Char Data Type

Definition: The char data type in C# is a single 16-bit Unicode character. It is used to store individual characters such as letters, digits, or symbols.
Size: 16 bits (2 bytes)
Range: The char type can represent any Unicode character from U+0000 to U+FFFF.
Examples:

```
char letter = 'A';
char digit = '1';
char symbol = '#';
```

Typical Use Cases:

- Storing individual characters.
- Handling characters in strings or text processing.
- Representing control characters (e.g., newline `\n`, tab `\t`).

String Data Type

Definition: The string data type in C# is a sequence of char objects. It is used to represent text as a series of characters.
Size: The size of a string is dynamic and depends on the number of characters it contains. Each character in the string uses 2 bytes.
Range: The length of a string is limited by system memory.
Examples:

```
string greeting = "Hello, World!";
string name = "Alice";
string empty = ""; // An empty string
```

Typical Use Cases:

- Storing and manipulating text data.
- User input and output.
- File I/O operations involving text.
- Formatting and displaying messages.

What is Unicode?

- Unicode is an international encoding standard for use with different languages and scripts. It aims to provide a unique number (code point) for every character, no matter the platform, program, or language.
- Unicode was created to address the limitations of earlier character encoding schemes, which often supported only a limited set of characters, leading to incompatibility issues.
- The Unicode standard includes characters from almost all writing systems, symbols, and control characters used in text processing.
- Characters in Unicode are identified by unique code points, written in the format U+XXXX, where XXXX is a hexadecimal number.

Practical Examples of char and string

- **Using char:**

```
char initial = 'J';
Console.WriteLine("Initial: " + initial); // Output: Initial: J
```

- **Using string:**

```
string message = "Welcome to C# programming!";
Console.WriteLine(message); // Output: Welcome to C# programming!
```

- **Manipulating strings:**

```
string firstName = "John";
string lastName = "Doe";
string fullName = firstName + " " + lastName;
Console.WriteLine("Full Name: " + fullName); // Output: Full Name:
   John Doe
```

- **Accessing characters in a string:**

```
string word = "Hello";
char firstLetter = word[0]; // 'H'
char lastLetter = word[word.Length - 1]; // 'o'
Console.WriteLine("First letter: " + firstLetter); // Output: First
    letter: H
Console.WriteLine("Last letter: " + lastLetter); // Output: Last
   letter: o
```

- **String methods:**

```
string phrase = " C# is fun! ";
string trimmedPhrase = phrase.Trim(); // "C# is fun!"
string upperPhrase = phrase.ToUpper(); // " C# IS FUN! "
bool containsWord = phrase.Contains("fun"); // true
Console.WriteLine("Trimmed: '" + trimmedPhrase + "'"); // Output: '
   C# is fun!'
Console.WriteLine("Uppercase: '" + upperPhrase + "'"); // Output: '
    C# IS FUN! '
Console.WriteLine("Contains 'fun': " + containsWord); // Output:
   Contains 'fun': True
```

Boolean Data Types in C#

The boolean data type in C# is fundamental for controlling the flow of a program and making decisions. Understanding how to use booleans effectively is essential for writing clear and logical code.

Definition of Boolean Data Type

The bool data type in C# represents a boolean value, which can be either true or false. This data type is named after the British mathematician and logician George Boole, who developed the algebraic system of logic that bears his name.
Size: 1 bit (theoretically) but typically occupies 1 byte in memory for alignment purposes.
Range: The bool type can only take on two values: true or false.
Syntax:

```
bool isTrue = true;
bool isFalse = false;
```

Typical Use Cases

- Booleans are primarily used in conditional statements to control the flow of execution.

```
bool isRaining = true;

if (isRaining)
{
    Console.WriteLine("Take an umbrella.");
}
else
{
    Console.WriteLine("Enjoy the sunshine!");
}
```

- Booleans are also used to determine whether loops should continue running.

```
bool keepRunning = true;
int counter = 0;

while (keepRunning)
{
    Console.WriteLine("Loop iteration: " + counter);
    counter++;
    if (counter >= 5)
    {
        keepRunning = false;
    }
}
```

- Booleans can sometimes serve as flags to indicate the state of an object or condition.

```
bool isGameOver = false;

// Game logic here

isGameOver = true; // Set the flag to true when the game is over
```

Understanding Value Types vs. Reference Types

In C#, all data types are categorized as either value types or reference types. This distinction determines how data is stored, accessed, and managed in memory.

Value Types

Value types are types that hold their data directly. When you assign a value type to a variable, the actual data is stored in that variable. Value types are usually stored in the stack memory, which is a region of memory that is highly efficient for managing data with a short lifespan.

Characteristics

- **Direct Storage:** The variable contains the actual data.
- **Memory Allocation:** Typically allocated on the stack.
- **Default Value:** Each value type has a default value (e.g., 0 for numeric types, false for bool).
- **Immutability in Assignments:** Assigning one value type variable to another copies the value, not the reference.

Examples of Value Types

int (Integer)

```
int count = 10;
```

`count` holds the actual integer value 10.

float (Floating-Point)

```
float temperature = 36.6f;
```

`temperature` holds the actual floating-point value 36.6.

bool (Boolean)

```
bool isCompleted = true;
```

`isCompleted` holds the actual boolean value true.

Reference Types

Reference types store references to their data, rather than the data itself. When you assign a reference type to a variable, the variable holds a reference (or address) pointing to the actual data stored in the heap memory. The heap is a region of memory used for dynamic memory allocation, suitable for data with a longer lifespan.

Characteristics

- **Indirect Storage:** The variable contains a reference to the actual data.
- **Memory Allocation:** Typically allocated on the heap.
- **Default Value:** The default value of a reference type is `null`, indicating that it does not refer to any object.
- **Mutability in Assignments:** Assigning one reference type variable to another copies the reference, not the actual data.

Examples of Reference Types

string

```
string message = "Hello, World!";
```

`message` holds a reference to the string data "Hello, World!" stored in the heap.

arrays

```
int[] numbers = { 1, 2, 3, 4, 5 };
```

`numbers` holds a reference to an array of integers stored in the heap.

objects

```
class Person
{
    public string Name { get; set; }
}
Person person = new Person();
person.Name = "Alice";
```

`person` holds a reference to an instance of the `Person` class stored in the heap.

Key Differences

Understanding how value types and reference types are stored and managed in memory is essential for efficient programming.

Memory Allocation

Value Types:

- Stored on the stack.
- Stack memory allocation is fast and involves a Last In, First Out (LIFO) strategy.
- Each variable has its own copy of the data.

Reference Types:

- Stored on the heap.
- Heap memory allocation is more complex and involves dynamic memory management.
- Variables hold references to the same data, allowing shared access.

Example of Value Types in Memory

```
int a = 5;
int b = a; // b is a copy of a, with its own storage
a = 10; // Changing a does not affect b
Console.WriteLine(b); // Output: 5
```

In this example, `a` and `b` are stored in the stack. `b` gets its own copy of 5, so changing `a` to 10 does not affect `b`.

Example of Reference Types in Memory

```
Person person1 = new Person();
person1.Name = "Alice";
Person person2 = person1; // person2 references the same Person object
    as person1
person1.Name = "Bob";
Console.WriteLine(person2.Name); // Output: Bob
```

In this example, `person1` and `person2` both reference the same `Person` object in the heap. Changing the `Name` property via `person1` affects `person2` as well because they refer to the same object.

Type Conversion and Casting

Type Conversion and Casting in C# are techniques used to convert a value from one data type to another. This is essential when working with different data types in a program, ensuring compatibility and proper manipulation of data.

Type Conversion

Type Conversion refers to the process of converting a value from one data type to another. This can be done implicitly or explicitly. Implicit conversion happens automatically when there is no risk of data loss or overflow, such as converting an integer to a float. Explicit conversion, also known as casting, is required when there is potential for data loss or when converting between incompatible types, and it requires specific syntax.

Casting

Casting is the technique of explicitly converting a value from one data type to another. This is necessary when the conversion is not inherently safe or straightforward, such as converting a float to an integer, which can result in loss of precision. Casting is done using a cast operator to instruct the compiler to convert the data type.

Implicit Conversion in C#

Implicit conversion refers to the automatic transformation of a value from one data type to another without requiring explicit syntax from the programmer. This type of conversion is only possible when there is no risk of data loss or overflow, ensuring that the operation is safe and meaningful.

Rules for Implicit Conversion

Implicit conversion can occur in the following scenarios:

- When converting from a smaller to a larger numeric type.
- When converting from a derived class to a base class.
- When converting between compatible data types where the conversion is guaranteed to be safe.

Examples of Implicit Conversion

Numeric Conversions:

- From `int` to `long`
- From `float` to `double`
- From `byte` to `int`

Reference Type Conversions:

- From a derived class to a base class

Example: Converting Smaller Numeric Types to Larger Numeric Types

```
int smallNumber = 123;
long largeNumber = smallNumber; // int to long conversion
Console.WriteLine(largeNumber); // Output: 123

float smallFloat = 12.34F;
double largeDouble = smallFloat; // float to double conversion
Console.WriteLine(largeDouble); // Output: 12.34
```

In this example, the integer `smallNumber` is implicitly converted to a `long`, and the float `smallFloat` is implicitly converted to a `double`. These conversions are safe because the target types (`long` and `double`) can represent all possible values of the source types (`int` and `float`).

Example: Implicit Conversion Between `char` and `int`

```
char letter = 'A';
int letterCode = letter; // char to int conversion
Console.WriteLine(letterCode); // Output: 65
```

In this example, the `char letter` is implicitly converted to an `int`. The Unicode code point of the character 'A' is 65, which is the value assigned to `letterCode`.

Key Points to Remember

- Implicit conversions are designed to prevent data loss or overflow. This is why they are limited to conversions that are guaranteed to be safe.
- Implicit conversions help simplify code and improve readability by removing the need for explicit cast syntax.
- The C# compiler handles implicit conversions automatically, allowing you to focus on the logic of your code without worrying about type compatibility issues.

When Implicit Conversion is Not Possible

Implicit conversion is not allowed when there is a possibility of data loss or when the types are not compatible. In such cases, explicit conversion (casting) is required.

Explicit Conversion

Explicit conversion (also known as casting) is the process of converting one data type to another with the use of a cast operator. Unlike implicit conversion, explicit conversion requires the programmer to specify the type conversion explicitly because there might be a risk of data loss or overflow. Explicit conversion is necessary when the types are not compatible for implicit conversion or when narrowing conversions are involved.

Syntax for Explicit Conversion

Explicit conversion requires the use of a cast operator to specify the target data type. This tells the compiler that you intend to convert the value to a different type, even though the conversion might not be inherently safe.

Syntax for Explicit Conversion:

```
targetType variableName = (targetType) value;
```

Examples of Explicit Conversion

Numeric Conversions:

- From a larger numeric type to a smaller numeric type.
- From a floating-point type to an integer type.

Reference Type Conversions:

- From a base class to a derived class (with potential runtime checks).

Example 1: Converting Larger Numeric Types to Smaller Numeric Types

```
int largeNumber = 123456;
short smallNumber = (short)largeNumber; // Explicit conversion
Console.WriteLine(smallNumber); // Output: -7616 (possible data loss
   due to overflow)
```

In this example, the integer `largeNumber` is explicitly converted to a `short`. Since `short` has a smaller range than `int`, this conversion might result in data loss or overflow, as seen in the unexpected output.

Example 2: Converting Floating-Point Types to Integer Types

```
double pi = 3.14159;
int wholePi = (int)pi; // Explicit conversion
Console.WriteLine(wholePi); // Output: 3 (fractional part is truncated)
```

In this example, the double `pi` is explicitly converted to an integer. The fractional part is truncated, and only the whole number part remains.

Key Considerations for Explicit Conversion

- **Data Loss:** Explicit conversions can lead to data loss, especially when converting from a larger type to a smaller type or from a floating-point type to an integer type.
- **Precision Loss:** When converting floating-point numbers to integers, the fractional part is lost.
- **Runtime Errors:** Explicit conversion between incompatible reference types can result in runtime errors. It's important to ensure that the conversion is valid.

Conversion Methods

In C#, the `Convert` class provides a set of static methods that facilitate type conversion between different data types. These methods are robust and help ensure that conversions are performed accurately and safely, handling potential issues like null values and incompatible types.

Using the Convert Class for Type Conversion

The `Convert` class in C# is part of the `System` namespace and provides a variety of methods to convert a base data type to another base data type. These methods are particularly useful when dealing with user input, data parsing, and other scenarios where data types may vary.

Common Methods in the Convert Class

- `ToBoolean`
- `ToByte`
- `ToChar`
- `ToDateTime`
- `ToDecimal`
- `ToDouble`
- `ToInt16`
- `ToInt32`
- `ToInt64`
- `ToString`

Each method attempts to convert the given value to the specified type, throwing exceptions if the conversion is not possible.

Examples of Conversion Methods

Example 1: Converting a String to an Integer

```
string numberString = "123";
int number = Convert.ToInt32(numberString);
Console.WriteLine(number); // Output: 123
```

In this example, the `Convert.ToInt32` method converts the string "123" to the integer 123. If the string cannot be converted to an integer (e.g., if it contains non-numeric characters), a `FormatException` will be thrown.

Example 2: Converting a String to a Double

```
string doubleString = "123.45";
double doubleNumber = Convert.ToDouble(doubleString);
Console.WriteLine(doubleNumber); // Output: 123.45
```

Here, the `Convert.ToDouble` method converts the string "123.45" to the double 123.45. The method handles both integer and decimal parts of the string.

Example 3: Converting a Boolean to a String

```
bool isTrue = true;
string boolString = Convert.ToString(isTrue);
Console.WriteLine(boolString); // Output: True
```

The `Convert.ToString` method converts the boolean value `true` to the string "True". This method is useful for displaying boolean values as strings in user interfaces or logs.

Example 4: Converting a String to a DateTime

```
string dateString = "2024-05-26";
DateTime date = Convert.ToDateTime(dateString);
Console.WriteLine(date); // Output: 5/26/2024 12:00:00 AM
```

In this example, the `Convert.ToDateTime` method converts the string "2024-05-26" to a `DateTime` object representing May 26, 2024. If the string format is not recognizable as a valid date, a `FormatException` will be thrown.

Example 5: Converting an Integer to a Byte

```
int intValue = 255;
byte byteValue = Convert.ToByte(intValue);
Console.WriteLine(byteValue); // Output: 255
```

The `Convert.ToByte` method converts the integer 255 to the byte 255. If the integer value is outside the range of a byte (0 to 255), an `OverflowException` will be thrown.

Parsing Methods

Parsing methods are used to convert strings into other data types. Unlike the `Convert` class methods, parsing methods are specifically designed to handle string inputs and convert them into corresponding data types. These methods are commonly found in various data types such as `int`, `double`, `bool`, `DateTime`, and others.

What is Parsing?

Parsing is the process of analyzing a string of characters to extract meaningful information and convert it into a specific data type. Parsing methods interpret the string according to the rules of the target data type and return the corresponding value.

Common Parsing Methods

- `int.Parse`
- `double.Parse`
- `bool.Parse`
- `DateTime.Parse`
- `Enum.Parse`

Examples of Parsing Methods

Example 1: Parsing a String to an Integer

```
string numberString = "123";
int number = int.Parse(numberString);
Console.WriteLine(number); // Output: 123
```

In this example, the `int.Parse` method converts the string "123" into the integer 123. If the string cannot be parsed into an integer (e.g., if it contains non-numeric characters), a `FormatException` will be thrown.

Example 2: Parsing a String to a Double

```
string doubleString = "123.45";
double doubleNumber = double.Parse(doubleString);
Console.WriteLine(doubleNumber); // Output: 123.45
```

Here, the `double.Parse` method converts the string "123.45" into the double 123.45. The method handles both the integer and fractional parts of the string.

Example 3: Parsing a String to a Boolean

```
string boolString = "true";
bool isTrue = bool.Parse(boolString);
Console.WriteLine(isTrue); // Output: True
```

In this example, the `bool.Parse` method converts the string "true" into the boolean value `true`. The method is case-insensitive and can handle strings like "true", "True", "false", and "False".

Example 4: Parsing a String to a DateTime

```
string dateString = "2024-05-26";
DateTime date = DateTime.Parse(dateString);
Console.WriteLine(date); // Output: 5/26/2024 12:00:00 AM
```

The `DateTime.Parse` method converts the string "2024-05-26" into a `DateTime` object representing May 26, 2024. The method can handle various date and time formats.

Example 5: Parsing a String to an Enum

```
enum Colors { Red, Green, Blue }

string colorString = "Green";
Colors color = (Colors)Enum.Parse(typeof(Colors), colorString);
Console.WriteLine(color); // Output: Green
```

In this example, the `Enum.Parse` method converts the string "Green" into the corresponding `Colors` enum value `Colors.Green`.

Key Points to Remember

- The string must be in a format that is valid for the target data type. If the string format is incorrect, a `FormatException` will be thrown.
- Parsing methods typically ignore leading and trailing whitespace in the string.
- For numeric and date/time parsing, the method may consider culture-specific formatting. By default, parsing methods use the current culture.