C# Data Types

Topics Covered

- The C# Type System
- Integral Types
- Floating-Point Types
- Other Built-In Types
- Aliases and Class Names
- Literals

- C# Operators
  - Arithmetic
  - Bitwise
  - Comparisons
  - Boolean
    - Short Circuiting

The C# Type System

C# (like all .NET languages) is a strongly-typed language. What does this mean?

The type of every variable is known when a program is compiled.

If there is a type mismatch in the program, it will not compile.

What is a type? Let's look at two primitive types, *int* and *double*.

Mathematically, an int is among the collection of positive and negative whole numbers denoted by $\mathbb{Z}$, numbers with no fractional part.

In contrast, a double represents a real number, represented by $\mathbb{R}$, which may have a fractional component out to an arbitrary number of decimal places.

The C# type system sees int and double as distinct and has strong rules about how they can be inter-converted.

platform By Per Scholas

Integral Types

Integral types represent whole numbers. They have no fractional part.

The C# language has 8 integral types (next slide).

They can be categorized by size (8, 16, 32 or 64 bits) and signed vs. unsigned.

The larger the size, the greater the range of values that can be represented.

The signed types can represent both negative and positive values.

The unsigned types can only represent zero and positive values.

C# Integral Types

C# Alias	.NET Class Name	Type	Size (bits)	Range (values)
byte	Byte	Unsigned integer	8	0 to 255
sbyte	SByte	Signed integer	8	-128 to 127
short	Int16	Signed integer	16	-32,768 to 32,767
ushort	UInt16	Unsigned integer	16	0 to 65,535
int	Int32	Signed integer	32	-2,147,483,648 to 2,147,483,647
uint	UInt32	Unsigned integer	32	0 to 4294967295
long	Int64	Signed integer	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	UInt64	Unsigned integer	64	0 to 18,446,744,073,709,551,615

Why So Many?

Why have so many types? Can’t we just always use good ol’ **int**?

There are two answers to this question. The first is about conserving memory.

For many programs, there is little concern about memory.

Some applications, such as a web service, need to be *scalable*. Scalability is the capability of an appication to perform well as it starts being used by increasing numbers of clients.

If you created a web service that performs complex calculations involving thousands of numbers and you knew, for example, that the numbers were always between -10,000 and 10,000, you would want to use the smallest numeric type that could represent that range (short).

We also need to be conscientious about memory usage when writing micro-processor applications.

Numbers for Problem Domains

Sometimes we use types that are appropriate to a problem domain.

For example, IP addresses (IPv4) are represented as values like 172.16.254.1.

Each of the 4 parts of the address can take on values from 0 to 255 – perfectly represented by a byte.

Another domain where the byte is perfect is the RGB color model used by CSS and other technologies. Here, a color is represented by mixtures of red, green and blue.

Each value or r, g and b can take on values from 0 to 255.

When we write code to process IPv4 addresses or RGB colors, we would use bytes.

Using a byte gives us another advantage: the compiler will prevent us from unintentionally assigning a value outside of the byte range.

C# Floating-Point Types

C# has just 3 floating-point types.

The decimal type has a smaller range and higher precision than double. It should be used for financial applications.

C# Alias	.NET Class Name	Type	Size (bits)	Range (values)
float	Single	Single-precision floating point type	32	$\pm 1.5 \text{ x } 10^{-45}$ to $\pm 3.4 \text{ x } 10^{38}$
double	Double	Double-precision floating point type	64	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$
decimal	Decimal	Precise fractional or integral type that can represent decimal numbers with 29 significant digits	128	$\pm 1.0 \text{ x } 10^{28}$ to $\pm 7.9 \text{ x } 10^{28}$

platform
By Per Scholas

Other Built-In Types

C# Alias	.NET Class Name	Type	Size (bits)	Range (values)
char	Char	A single Unicode character	16	Unicode symbols used in text
bool	Boolean	Logical Boolean type	8	true or false
object	Object	Base type of all other types		
string	String	A sequence of characters		
DateTime	DateTime	Represents date and time	64	0:00:00am 1/1/01 to 11:59:59pm 12/31/9999

C# Aliases and Class Names

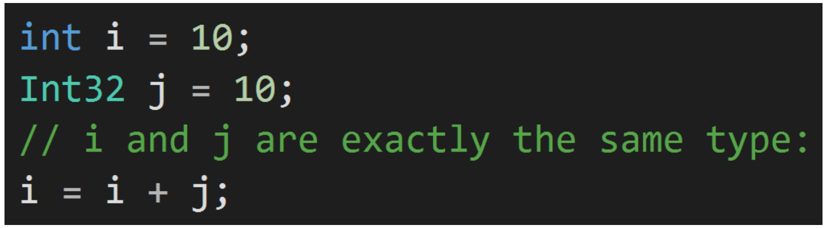
From the previous tables, what is the difference between an alias and the .NET class name?

We mentioned previously that every variable in C# is an Object, even numbers.

Numbers are Objects and they have methods (such as double.Parse), and we've learned that methods can only be defined in the context of a class.

Therefore: numbers are classes.

We have the option of declaring our numeric variables using their full class names, but using the aliases is usually simpler and easier to understand.



```
int i = 10;
Int32 j = 10;
// i and j are exactly the same type:
i = i + j;
```

platform
By Per Scholas

Numeric, char, and bool Literals

Most often we can type literal numbers into our code and the compiler understands their type based on context.

Occasionally we need to tell the compiler that a number is a specific type which is not evident from context.

```
int i = 55876;      // int literal
uint u = 55876u;    // uint literal
long l = 55876L;    // long literal
ulong ul = 55876ul; // ulong literal
i = 0x2847;         // int hexadecimal
u = 0x2847u;        // uint hex
l = 0x2847L;        // long hex
ul = 0x2847ul;      // ulong hex
```

```
double d = 3.14159;      // double literal
float f = 3.14159f;      // float literal
decimal dec = 3.14159M;  // decimal literal
d = 6.023e-23;           // double scientific notation
f = 6.023e-23f;          // float sci
dec = 6.023e-23M;        // decimal sci
```

```
bool t = true;
bool f = false;
```

```
char c = 'a';          // char literal
c = '\t';             // char literal using escape
c = '\u1f60';         // char literal unicode
```

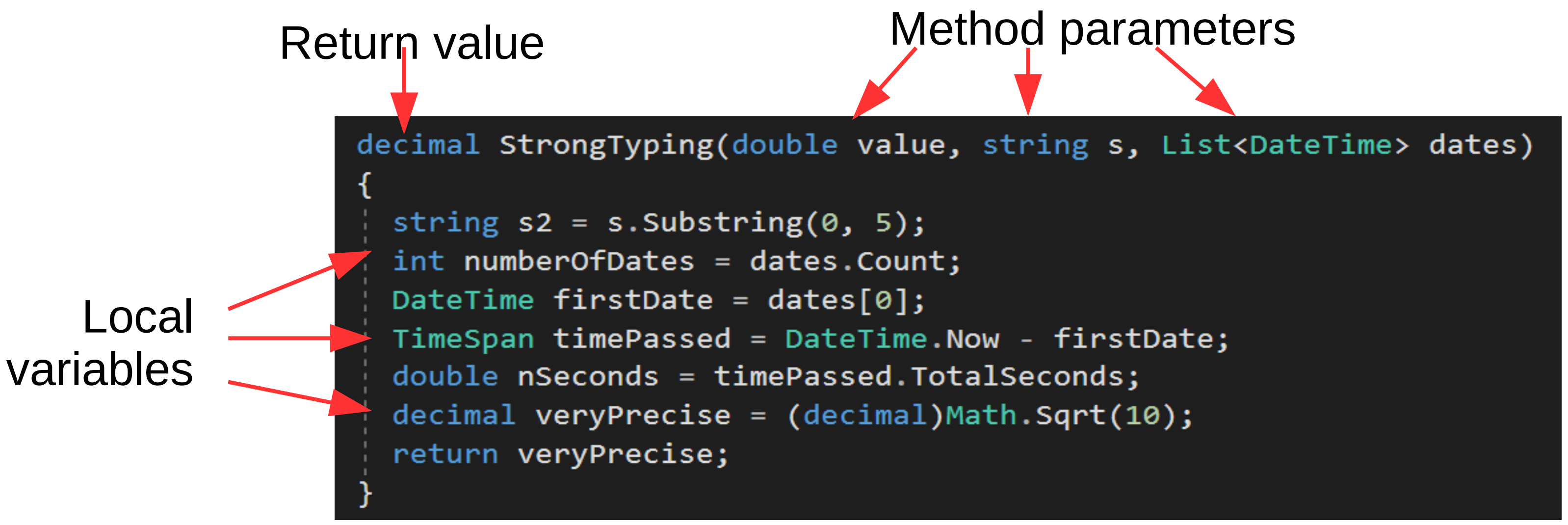
platform
By Per Scholas

C# is Strongly Typed

The type of every variable is known exactly – **<u>no exceptions</u>**!

This is true <u>everywhere</u> - parameters, local variables, return values

Every variable is declared: <typename> <variable name> [initialization];



```
decimal StrongTyping(double value, string s, List<DateTime> dates)
{
  string s2 = s.Substring(0, 5);
  int numberOfDates = dates.Count;
  DateTime firstDate = dates[0];
  TimeSpan timePassed = DateTime.Now - firstDate;
  double nSeconds = timePassed.TotalSeconds;
  decimal veryPrecise = (decimal)Math.Sqrt(10);
  return veryPrecise;
}
```

C# Operators

Operators are program elements, most often single-character symbols, which can be applied to *operands*.

Operators can be classified into three groups based on their *arity*. C# has unary, binary, and ternary operators.

Some operators are arithmetic and work with numeric operands: + - * / %

Some operators work with numeric operands and return bool: < > <= >= !=

Some operators are bitwise and work with integral operands: ~ & ^ | << >>

Some operators are logic and work with boolean operands: && || == !=

The **new** operator allocates memory to create an object.

An exhaustive description of the C# operators is found here.

Arithmetic Operators

These examples use **double**, but the arithmetic operators can also be applied to integer types.

Be careful with integer division!

The increment and decrement operators can also be applied to floating-point types.

```
double d1 = 10, d2 = 20;
double d3 = d1 + d2;              // + operator
double d4 = d1 - d2;              // - operator
d1 = d2 * d3;                     // * operator (multiplication)
d2 = d3 / d4;                     // / operator (subtraction)
d1 = d2 % d3;                     // modulus (remainder)
```

```
d1 = d1 + d2;                     // Same outcome as next line
d1 += d2;                         // +=    add and assign.
d1 /= d2;                         // /=    divide and assign.
d1 *= d2;  d1 -= d2;              // etc.
```

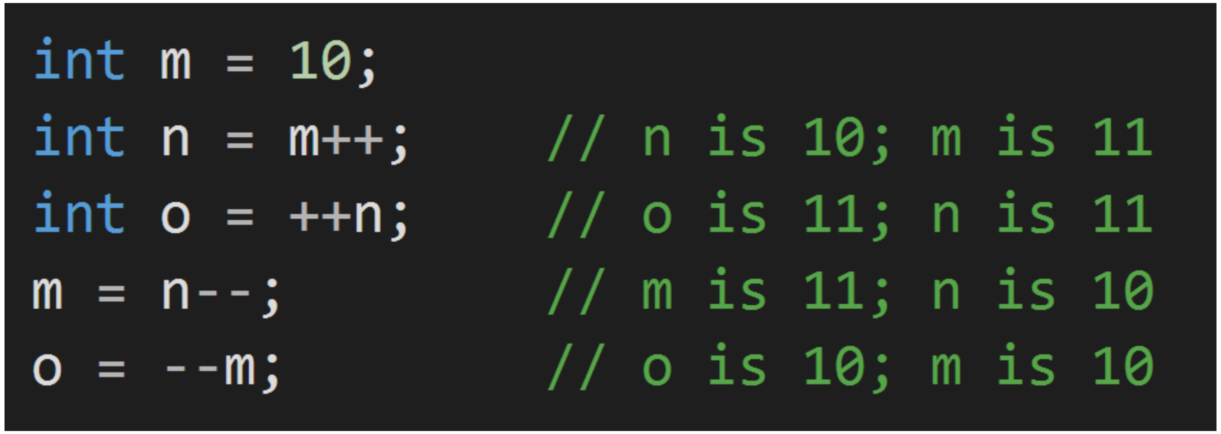
```
int i = 5, j = 7;
++i;                              // prefix increment
i++;                              // postfix increment
--i; i--;                         // prefix/postfix decrement
```

platform
By Per Scholas

Postfix and Prefix Behavior

The ++ and -- operators increase or decrease the value of an integer by 1.

They can be used in either prefix or postfix positions as shown here:



```
int m = 10;
int n = m++;    // n is 10; m is 11
int o = ++n;    // o is 11; n is 11
m = n--;        // m is 11; n is 10
o = --m;        // o is 10; m is 10
```

In postfix position, the assignments are made <u>before</u> the operator is applied.

In prefix position, the assignments are made <u>after</u> the operator is applied.

Integer Division

The “division” we’ve all learned is floating-point division. ½ = 0.5, of course!

When both operands of the / operator are both integer types, C# (like most strongly-typed computer languages) performs **integer division**.

The result of integer division is always an integer. Any fractional part that you might expect from a floating-point division is truncated. A few examples:

Expression	Result
1/2	0
3/2	1
8/3	2
13/4	3

The Modulus Operator

The modulus operator (%) returns the remainder of an integer division. A few examples:

Expression	Result
12 % 5	2
6 % 7	6
22 % 3	1
8 % 2	0

One of the most common uses of % is to test for even/odd values:

```
int testVal = 13;
bool isEven = testVal % 2 == 0;    // false
bool isOdd = testVal % 2 == 1;     // true
```

Algebraic Expressions

Developers occasionally need to translate algebraic expressions to code. Here is an example:

$$\frac{3+4x}{5} - \frac{10(y-5)(a+b+c)}{x} + 9(\frac{4}{x} + \frac{9+x}{y})$$

The equivalent in C# code is this:

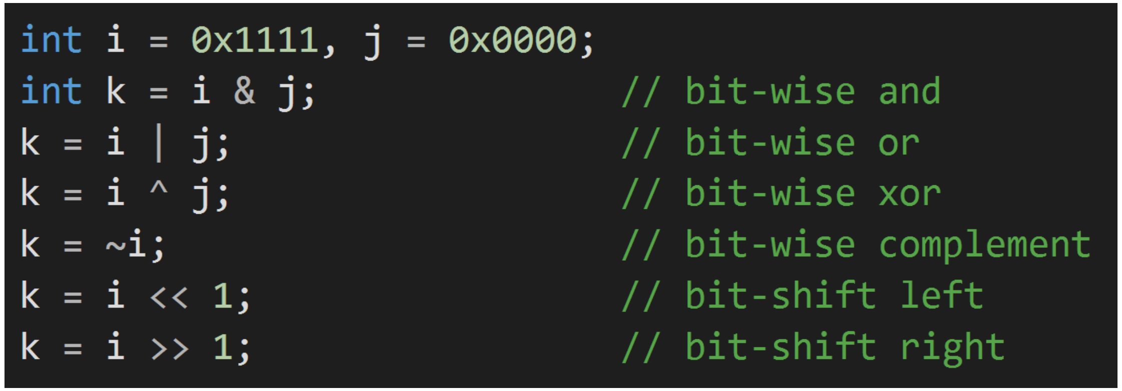
```
(3+4*x)/5 – 10*(y-5)*(a+b+c)/x + 9*(4/x + (9+x)/y)
```

Notice the use of operator precedence and parentheses.

Bitwise Operators

Left Operand	Right Operand	Operator	Result
11001100	01010101	& (AND)	01000100
11001100	01010101	| (OR)	11011101
11001100	01010101	^ (XOR)	10011001
	01010101	~ (NOT)	10101010
11001100	2	<< (shift left)	00110000
11001100	2	>> (shift right)	00110011

Bitwise Operators - Usage



```
int i = 0x1111, j = 0x0000;
int k = i & j;                // bit-wise and
k = i | j;                    // bit-wise or
k = i ^ j;                    // bit-wise xor
k = ~i;                       // bit-wise complement
k = i << 1;                   // bit-shift left
k = i >> 1;                   // bit-shift right
```

The bitwise operators manipulate the bits of integer types.

Bitwise operators are used in a technique called bit-masking.

Example: a long variable can represent 64 true/false values. To achieve the same with bool, we would need to use 8-times as much memory.

Bit operations are native to the CPU and are fast.

Comparison Operators

The comparison operators accept numeric operands and return a boolean value. They compare one number to another.

Equality/inequality comparisons betweens integer types are exact and predictable.

These comparisons between floating-point types should be avoided - they are susceptible to rounding errors.

They are better implemented by testing for equality within a difference.

Operator	Example	Result
Greater Than	12 > 3	true
Greater Than or Equal To	3 >= 5	false
Less Than	12 < 3	false
Less Than or Equal To	3 <= 5	true
Equal To	8 == 8	true
Not Equal To	8 != 8	false

Boolean Operators

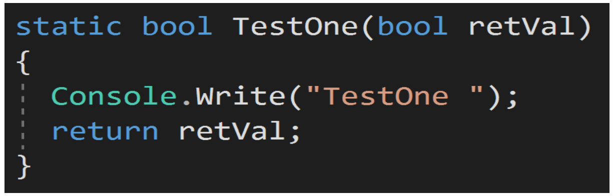
The boolean operators apply boolean logic to boolean operands. Examples:

Operator	Example	Result	Comments
Conditional AND	(true && false)	false	Evaluation of right operand short-circuits
Conditional OR	(true || false)	true	Evaluation of right operand short-circuits
Logical AND	(true & false)	false	Does not short-circuit
Logical OR	(true | false)	true	Does not short-circuit
Logical XOR	(true ^ false)	true	Does not short-circuit

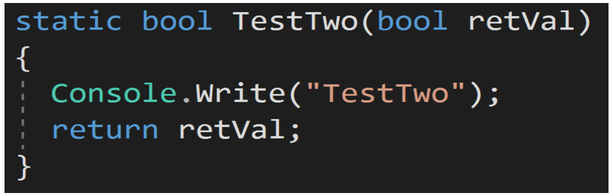
platform
By Per Scholas

Boolean Short-Circuiting

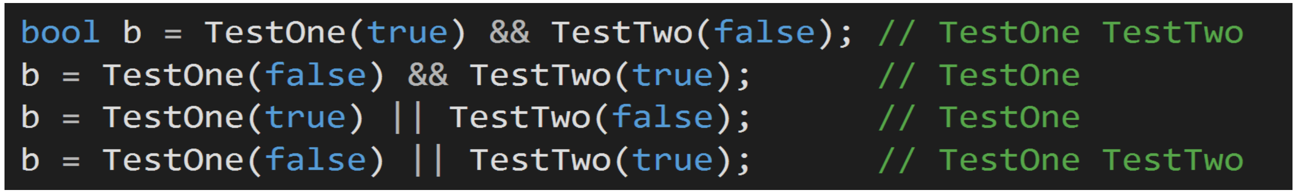
These code snippets illustrate short-circuiting of the boolean conditional operators:



```
static bool TestOne(bool retVal)
{
    Console.Write("TestOne ");
    return retVal;
}
```



```
static bool TestTwo(bool retVal)
{
    Console.Write("TestTwo");
    return retVal;
}
```



```
bool b = TestOne(true) && TestTwo(false); // TestOne TestTwo
b = TestOne(false) && TestTwo(true);      // TestOne
b = TestOne(true) || TestTwo(false);      // TestOne
b = TestOne(false) || TestTwo(true);      // TestOne TestTwo
```

If the left operand of **&&** evaluates false, then there is no need to evaluate the right operand.

If the left operand of **||** evaluates true, then there is no need to evaluate the right operand.

We'll see soon how this behavior can help us write concise and optimized code.

Topics Covered

- The C# Type System
- Integral Types
- Floating-Point Types
- Other Built-In Types
- Aliases and Class Names
- Literals

- C# Operators
  - Arithmetic
  - Bitwise
  - Comparisons
  - Boolean
    - Short Circuiting

Questions

- What is a *type*? Give examples. How are they different?
- C# has 8 different built-in integral types. How are they different. How do we decide which to use in a program?
- What are the C# floating-point types?
- Why do the built-in types have both aliases and class names?
- What is a numeric literal? How do you tell the compiler that a literal number is, for example, an unsigned integer?
- What does it mean if a language is *strongly typed*?
- What is the difference between prefix and postfix operators?
- What is operator short-circuiting?

Exercises

- Write an application to convert temperatures between fahrenheit (F) and celsius (C).
- The program operates in either of two modes: F → C or C → F.
- The program lets the user switch between these modes.
- The program allows the user to convert as many temperatures as they need, then quit when they are finished.

- The formula for F → C is: $C = \frac{5}{9}(F - 32)$

- The formula for C → F is: $F = \frac{9}{5}C + 32$