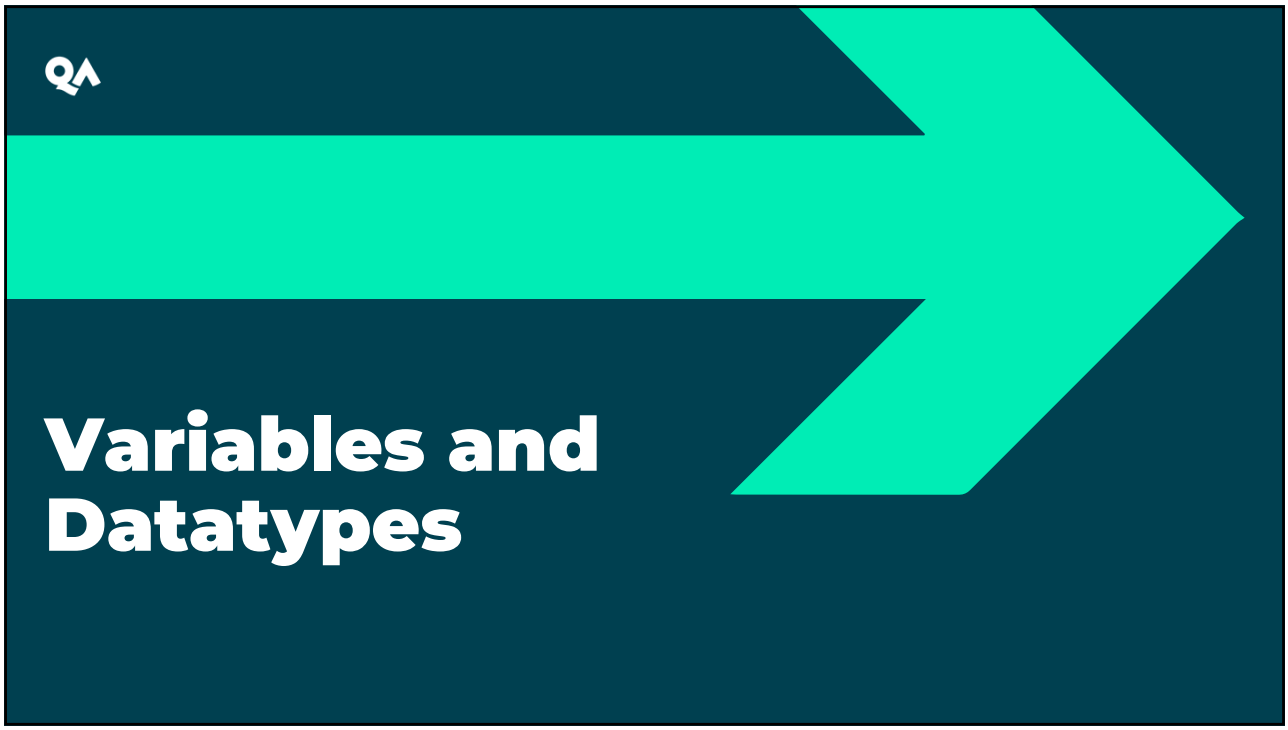
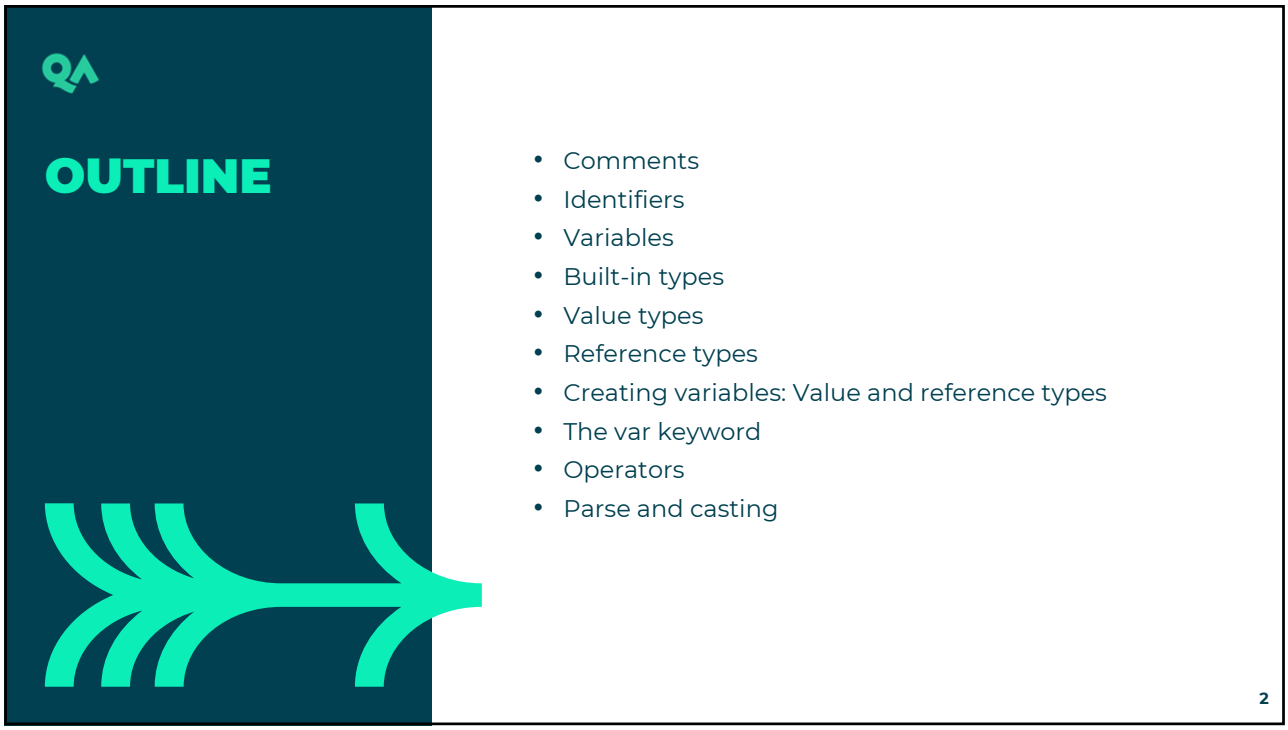

Variables and
Datatypes

1


OUTLINE
Comments
Identifiers
Variables
Built-in types
Value types
Reference types
Creating variables: Value and reference types
The var keyword
Operators
Parse and casting
2

2

COMMENTS

C# supports three styles of comments:

- Block comments
- In-line comments
- XML documentation comments

```
/*
This is a block comment ...
*/
```

```
... // This is a rest-of-line comment
```

```
/// <summary>
/// This is an XML-based comment
/// There are compiler tags
/// And .NET documentation convention tags
/// </summary>
```

3

3

IDENTIFIERS

Identifiers are used for the names of types e.g., classes and variables.

- Start with a letter of the alphabet (or an underscore)
  - Subsequent characters can include numeric digits and underscores
- C# is case sensitive
  - Therefore two identifiers can be differentiated by case alone
  - 'speed' & 'Speed' are different
- Follow convention for the casing of identifiers
  - camelCasing - local variables, parameters, & private fields
  - PascalCasing - types and everything they 'expose'

4

4

VARIABLES

- A variable is a symbolic name for an address in memory
- A variable is a value that can change
- All variables must be declared before they are used
- A variable must also be initialised before being read
- Local variables (variables declared within a method) have no initial value

```
int myAge;
bool answer = true;
string myName = "Michael Caine";
int i = 0, j = 1;

myAge = 21;
```

5

5

BUILT-IN DATATYPES

There are two main categories of types in C#:

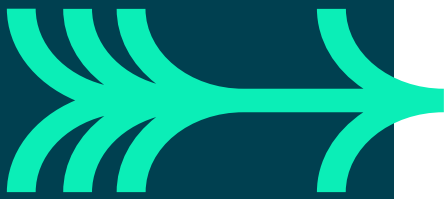
- Value types
- Reference types

- A variable of a **value** type contains an instance of the type
- A variable of a **reference** type contains a *reference* to an instance of the type
- With **reference** types, it is possible for two variables to reference the same object and for operations on one variable to affect the object referenced by another variable
- With **value** types, each variable has its own copy of data so it is not possible for operations on one to affect the other

6

6

VALUE TYPES VERSUS REFERENCE TYPES

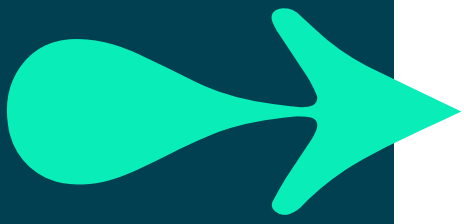


- A **value** type exists mainly for the data it holds rather than functionality
- A **reference** type is created for more complex types that need to exhibit a lot of functionality as well as store data
- The memory used by a **value** type is allocated in an area called the *stack*
- This memory is reclaimed as soon as the value type is no longer needed
- The memory used by a **reference** type is allocated in an area called the *managed heap*
- This memory is reclaimed by a service called the *Garbage Collector* - this happens at some future point when the object is no longer needed

7

7

BUILT-IN VALUE TYPES



C# has the following built-in value types, also known as *simple* types:

- Integral numeric types
- Floating-point numeric types
- bool
- char

A value type is one of the following:

- A **structure** type which encapsulates data and related functionality
- An **enumeration** type, which is a set of named constants

8

8

INTEGRAL NUMERIC TYPES

C# type/keyword	Range	Size	.NET type
`sbyte`	-128 to 127	Signed 8-bit integer	System.SByte
`byte`	0 to 255	Unsigned 8-bit integer	System.Byte
`short`	-32,768 to 32,767	Signed 16-bit integer	System.Int16
`ushort`	0 to 65,535	Unsigned 16-bit integer	System.UInt16
`int`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
`uint`	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
`ulong`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64
`nint`	Depends on platform	Signed 32-bit or 64-bit integer	System.IntPtr
`nuint`	Depends on platform	Unsigned 32-bit or 64-bit integer	System.UIntPtr

```
int x = 1234;
System.Int32 y = 1234;
```

9

9

FLOATING POINT NUMERIC TYPES

C# type/keyword	Approximate range	Precision	Size	.NET type
`float`	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~6-9 digits	4 bytes	System.Single
`double`	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits	8 bytes	System.Double
`decimal`	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits	16 bytes	System.Decimal

```
double x = 12.34;
System.Double y = 12.34;
```

```
float f = 12345.67;
```

readonly struct System.Double
Represents a double-precision floating-point number.

CS0664: Literal of type double cannot be implicitly converted to type 'float'; use an 'F' suffix to create a literal of this type

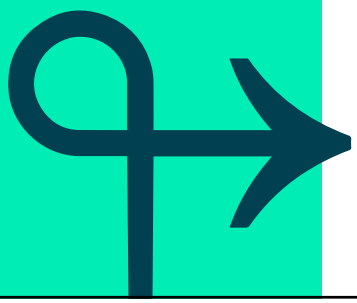
```
float f = 12345.67F;
double d = 12345.67;
double dd = 12345.67D;
decimal m = 12345.67M;
```

```
Console.WriteLine(double.NaN);
Console.WriteLine(double.NegativeInfinity);
Console.WriteLine(double.PositiveInfinity);
```

10

10

BOOL DATATYPE



The bool type can hold only one of **two** values: *true* or *false:*

```
bool check = true;
bool isValid = false;
```

The bool type is the result of **comparison** and **equality** operators:

```
Console.WriteLine(7.0 < 5.1);    // output: False
Console.WriteLine(5.1 > 5.1);    // output: False
Console.WriteLine(5.1 >= 5.1);    // output: True
Console.WriteLine(3.4 == 3.4);    // output: True
Console.WriteLine(3.4 != 3.4);    // output: False
Console.WriteLine(double.NaN < 7.0);    // output: False
```

11

11

CHAR DATATYPE

The char type is an alias for **System.Char**

It is a structure type that represents a Unicode 16-bit character.

A char can be specified with:

- A character literal in single quotes
- A Unicode escape sequence, which is **\u** followed by the 4-symbol hex character code
- A hexadecimal escape sequence, which is **\x** followed by the hex character code (leading zeros can be omitted)

```
char letter = 'p';
char copyrightUni = '\u00a9';
char copyrightHex = '\xa9';
char atSymbol = '\x40';
```

12

12

NULLABLE VALUE TYPES

A nullable value type **T?** represents all values of its underlying value type **T** and an additional *null* value.

For example, the nullable bool type can hold only one of **three** values: *true* or *false* or *null:*

```
bool? check = true;
bool? isValid = false;
bool? flag = null;
```

```
double? pi = 3.14159;
char? letter = 'p';

int luckyNumber = 7;
int? myLuckyNumber = luckyNumber;
```

13

13

STRUCTURE TYPES

A structure type (**struct**) is a value type that can encapsulate data and related functionality.

Use structs to design small data-centric types that provide little or no behaviour.

Microsoft recommend you define **immutable** structure types.

- Define the struct as *readonly*
- Define the data members as *readonly* or use an *init* accessor

14

14

STRUCT EXAMPLE

```
1 reference
public readonly struct Coords
{
    // Constructor
    // It is called when an instance is created and is used to initialize the instance
    0 references
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    // Auto-implemented properties: X and Y
    // A getter enables a property value to be read
    // An init-only setter assigns a value to the property only during object construction
    1 reference
    public double X { get; init; }
    1 reference
    public double Y { get; init; }

}
```

```
Coords c1 = new Coords(5, 7);
Coords c2 = new(5, 7);
Console.WriteLine(c1.X);
Console.WriteLine(c2.Y);
// c1.X = 10; // compile error because X is readonly
Console.WriteLine(c2.ToString());
```

15

15

ENUM TYPES

An enumeration type (**enum**) is a value type defined by a set of named constants of the underlying integral numeric type.

```
enum Level
{
    Low,    // 0
    Medium, // 1
    High    // 2
}
```

```
Level myVar = Level.Medium;
Console.WriteLine(myVar);
```

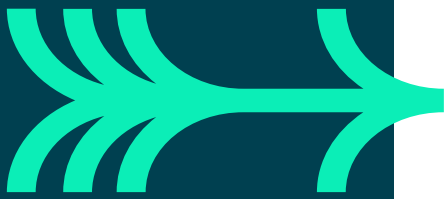
```
enum FiscalMonths
{

    April = 1,
    May = 2,
    June = 3,
    July = 4,
    August = 5,
    September = 6,
    October = 7,
    November = 8,
    December = 9,
    January = 10,
    February = 11,
    March = 12
}
```

```
FiscalMonths firstMonth = FiscalMonths.April;
Console.WriteLine(firstMonth);
```

16

16

REFERENCE TYPES



C# has the following built-in reference types:

- object
- string
- dynamic

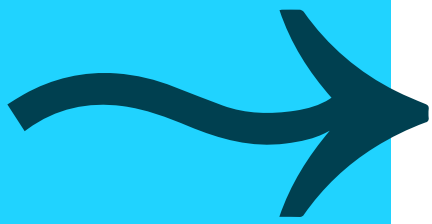
The following keywords are used to declare reference types:

- class
- interface
- delegate
- record

17

17

OBJECT



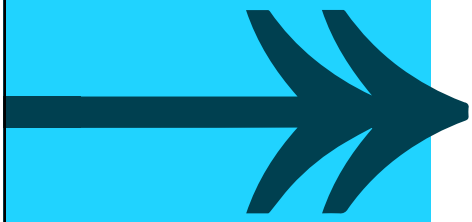
- **System.Object** is the ultimate base class of all .NET classes
- Inheritance from Object is implicit
- Every method defined in Object is available in all objects in the system
- Derived classes can override some of these methods:
  - **Equals**: Supports comparisons between objects
  - **Finalize**: Performs clean-up operations before an object is reclaimed
  - **GetHashCode**: Generates a number corresponding to the value of the object to support the use of a hash table
  - **ToString**: Manufactures a human-readable text string that describes an instance of the class
- When a variable of a value type is converted to object, it is said to be *boxed*
- When a variable of type object is converted to a value type, it is said to be *unboxed*

18

18

STRING

- A **string** type represents a sequence of zero or more Unicode characters
- string is an alias for **System.String**
- String is a reference type but the equality operators **==** and **!=** are defined to compare the *values* of string objects rather than their *references*
- The **+** operator concatenates strings
- Strings are immutable



```
string greeting = "good morning";
string message = "good ";
message = message + "morning";
Console.WriteLine(greeting == message);
Console.WriteLine(object.ReferenceEquals(greeting, message));
```

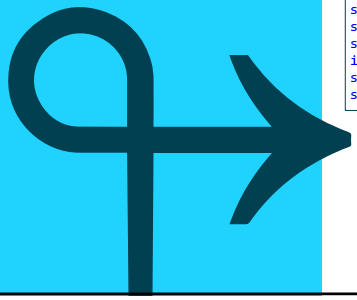
```
True
False
```

19

19

STRING LITERALS

- A string literal is anything between a pair of double quotes **“ “**
- A *verbatim* string allows special characters to be included in a string literal and is prefixed with **@**
- Use the **string.format** method to format strings or use *interpolated* strings
- Interpolated strings are prefixed with **$**



```
string literalString = "This is a string literal";
string verbatimString = @"This is a verbatim string and can contain special characters such as \ and \n";
string newLine = "\n";
string name = "Bart";
int age = 8;
string formatString = string.Format("My name is {0} and I am {1} years old", name, age.ToString());
string interpolatedString = $"My name is {name} and I am {age} years old";
```

```
This is a string literal
This is a verbatim string and can contain special characters such as \ and \n


My name is Bart and I am 8 years old
My name is Bart and I am 8 years old
```

20

20

CLASS

```
public class Car : Object
{
    const int NumWheels = 4;
    int cylinders;

    int Cylinders {
        get { return cylinders; }
        set { cylinders = value;}
    }
    public void Start()
    {
        // code in here
    }
    public event EventHandler AlarmDisabled;
}
```

All classes inherit from System.Object

A constant, immutable field

A field (a class-level variable)

A property with a getter and a setter

A method

An event

21

OBJECT INSTANTIATION

Class definition with a method:

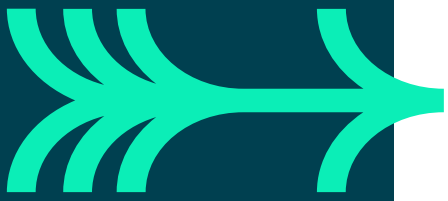
```
class Car
{
    2 references
    public void Register(string name, string address, string postCode, string country)
    {
        // store this in the DVLA database in Swansea
    }
}
```

Instantiate objects based on the class and invoke the method:

```
Car julieCar = new Car();
julieCar.Register("Julie Dooley", "1 Main Street", "CH12 9DL", "UK");


Car lisaCar = new Car();
lisaCar.Register("Lisa Simpson", "742 Evergreen Terrace", "97394", "USA");
```

22

CREATING VARIABLES: SIMPLE TYPES



Simple types are declared as variables using the following pseudo-code:

datatype variableName = value;

```
int a = 6; // literal assignment
float b = 7.5F; // literal assignment with a suffix
decimal c = 9.99M; // M or m suffix denotes a decimal literal
bool d = false; // assign special value false (or true) to a bool
char e = 'a'; // single quotes for char literals
char? h = null; // ? for a nulable variable
string f = "hello"; // string is a reference type but gets created like a value type
string? g = null; // nullable string
```

A value type holds its data within its own memory space.

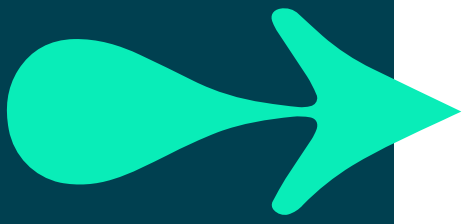
The **var** keyword can be used to implicitly type a variable.

```
int x = 3;
var y = 9;

Console.WriteLine(x.GetType()); // System.Int32
Console.WriteLine(y.GetType()); // System.Int32
```

23

23

CREATING VARIABLES: NON-SIMPLE VALUE TYPES



Non-simple value types are declared as variables using the '**new**' keyword for *struct* types and typically a value from the enumeration for an *enum* type:

```
Level level = Level.High;// enum variable is constrained to the defined values
Coords c1 = new Coords(b, a);// a struct can be instantiated using 'new'
Console.WriteLine(c1.GetType());// GetType method gets the type of the current instance
var c2 = new Coords(b, a); // the type can be inferred by the compiler when using 'var'
Console.WriteLine(c2.GetType());
Coords c3 = new(b, a); // C# 9 target-typed constructor invocation syntax
Console.WriteLine(c3.GetType()); // c1, c2 and c3 are all of type Coords
```

C# 9 introduced a shorter syntax for invoking a constructor called *target-typed invocation*.

```
Coords c3 = new(b, a); // C# 9 target-typed constructor invocation syntax
```

24

24

CREATING VARIABLES: REFERENCE TYPES

Reference types are instantiated as variables using the '**new**' keyword:

```
Car car1 = new Car(); // reference types are instantiated using 'new'
var car2 = new Car(); // the type can be inferred by the compiler
Car car3 = new(); // C# 9 target-typed constructor invocation syntax
Console.WriteLine(car3.GetType());
// 'if' is a conditional statement with the condition to be evaluated in brackets
if (car3 is Car) {
    // The 'is' operator checks if the result of an expression is compatible with a given type
    Console.WriteLine(nameof(car3) + " is a Car instance");
    // nameof produces the name of a variable, type, or member as a string constant
}
```

Target-typed constructor invocation can be used with any value or reference type that has a constructor.

A reference type holds a pointer (reference) within its own memory space that points to another memory location that holds the real data.

25

25

VARIABLE SCOPE

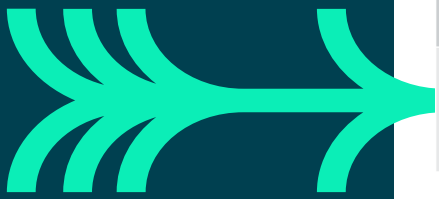
```
0 references
class ScopeOfVariables
{
    int visibleToEntireType; // variables at class scope (Fields)

    0 references
    int MethodA()
    {
        visibleToEntireType = 42;
        int onlyVisibleWithinMethod = 13;
        // block created with braces
        {
            int onlyVisibleWithinBlock = 5;
            return onlyVisibleWithinBlock + onlyVisibleWithinMethod;
        }

        onlyVisibleWithinBlock = 7; //not visible outside of the block (compile error)
    }

    0 references
    void MethodB()
    {
        visibleToEntireType = 21;
        onlyVisibleWithinMethod = 50; //not visible outside of MethodA (compile error)
    }
}
```

26

26

ACCESS MODIFIERS

Access Modifier	Description
public	The type or member can be accessed anywhere
private	The type or member can be accessed only by code in the same class or struct
protected	The type or member can be accessed only by code in the same class, or in a class that is derived from that class
internal	The type or member can be accessed by any code in the same assembly
protected internal	The type or member can be accessed by any code in the same assembly, or within a derived class in another assembly
private protected	The type or member can be accessed by types derived from the class that are declared within its containing assembly



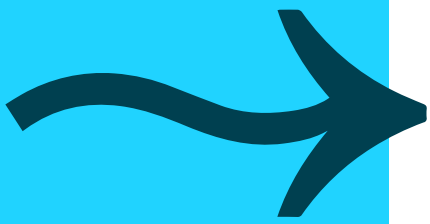
27

27

OPERATORS

C# provides many operators, such as:

- Arithmetic operators
- Comparison operators
- Boolean logical operators
- Bitwise and shift operators
- Equality operators



28

28

ARITHMETIC OPERATORS

- C# provides standard arithmetic operators:

++	**increment by 1**
--	**decrement by 1**
+	**plus**
-	**minus (numeric negation)**
+	**addition**
-	**subtraction**
*	**multiplication**
/	**division**
%	**modulo division (remainder)**

- Operators can use compound syntax:
- x = x + 5;
- x += 5;
- The above statements produce identical results

29

29

ARITHMETIC OPERATOR EXAMPLES: ++ --

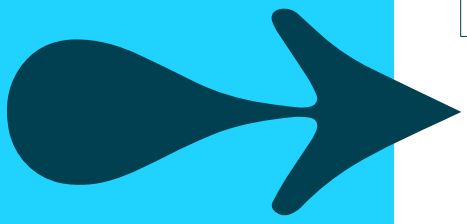
```
// post-fix increment operator
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);   // output: 4

// pre-fix increment operator
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);   // output: 2.5
```

```
// post-fix decrement operator
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);   // output: 2

// pre-fix decrement operator
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);   // output: 0.5
```

30

30

ARITHMETIC OPERATOR EXAMPLES: UNARY + -

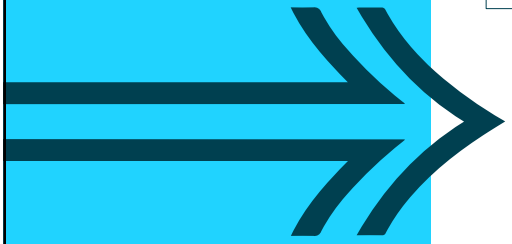


```
// unary plus and minus operators
Console.WriteLine(+8);     // output: 8

Console.WriteLine(-8);     // output: -8
Console.WriteLine(-(-8));  // output: 8

uint a = 9;
var b = -a;
Console.WriteLine(b);            // output: -9
Console.WriteLine(b.GetType());  // output: System.Int64

Console.WriteLine(-double.NaN);  // output: NaN
```

31

31

ARITHMETIC OPERATOR EXAMPLES: ADD + SUBTRACT -

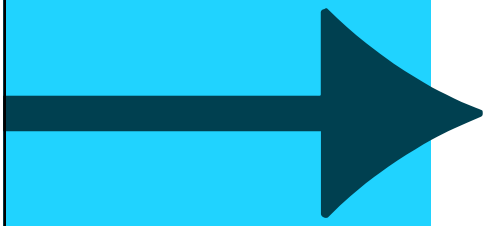


```
// addition operator
Console.WriteLine(5 + 4);       // output: 9
Console.WriteLine(5 + 4.3);     // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3

// subtraction operator
Console.WriteLine(47 - 3);      // output: 44
Console.WriteLine(5 - 4.3);     // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

32

32

ARITHMETIC OPERATOR EXAMPLES: MULTIPLY * DIVIDE /

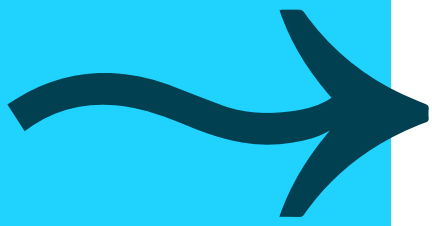


```
// multiply operator
Console.WriteLine(5 * 2);         // output: 10
Console.WriteLine(0.5 * 2.5);     // output: 1.25
Console.WriteLine(0.1m * 23.4m);  // output: 2.34

// division operator
// For integer operands, the result is an int type rounded towards zero
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2

// int / double
Console.WriteLine(13 / 5.0);       // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b);  // output: 2.6

Console.WriteLine(16.8f / 4.1f);   // output: 4.097561
Console.WriteLine(16.8d / 4.1d);   // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);   // output: 4.0975609756097560975609756098
```

33

33

ARITHMETIC OPERATOR EXAMPLES: REMAINDER /



```
// modulus / remainder operator
Console.WriteLine(5 % 4);   // output: 1
Console.WriteLine(5 % -4);  // output: 1
Console.WriteLine(-5 % 4);  // output: -1
Console.WriteLine(-5 % -4); // output: -1

Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m);  // output: 2.8
```

34

34

ARITHMETIC OPERATOR PRECEDENCE

- What are the values of *d*?

```
double a = 3;
double b = 5;
double c = 7;
double d = a + b * c;
```

```
double a = 3;
double b = 5;
double c = 7;
double d = (a + b) * c;
```

35

COMPARISON OPERATORS

- C# provides standard comparison operators:

>	**greater than**
>=	**greater than or equal to**
<	**less than**
<=	**less than or equal to**

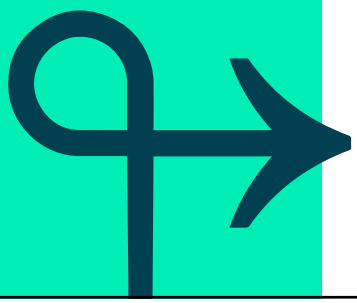
Comparison operators can be used to compare:

- All integral types
- All floating-point types
- char types based on the underlying character codes
- Enums based on the underlying integral values

36

36

COMPARISON OPERATOR EXAMPLES:



```
// less than <
Console.WriteLine(7.0 < 5.1);   // output: False
Console.WriteLine(5.1 < 5.1);   // output: False
Console.WriteLine(0.0 < 5.1);   // output: True


// greater than >
Console.WriteLine(7.0 > 5.1);   // output: True
Console.WriteLine(5.1 > 5.1);   // output: False

// less than equals <=  // greater than equals >=
Console.WriteLine(7.0 <= 5.1);   // output: False
Console.WriteLine(5.1 <= 5.1);   // output: True
Console.WriteLine(7.0 >= 5.1);   // output: True
Console.WriteLine(5.1 >= 5.1);   // output: True

// NaN
Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False

// char comparison
char a = 'a';
char b = 'b';
Console.WriteLine(a > b);   // output: False
Console.WriteLine(a <= b);  // output: True

// enum comparison
Level low = Level.Low;
Level high = Level.High;
Console.WriteLine(low > high);  // output: False
Console.WriteLine(low < high);  // output: True
```

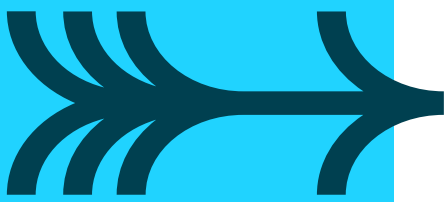
37

37

BOOLEAN LOGICAL OPERATORS

- C# provides standard Boolean logic operators:

!	**Logical negation**
&	**Logical AND**
|	**Logical OR**
^	**Logical exclusive OR**
&&	**Conditional Logic AND**
||	**Conditional Logic OR**

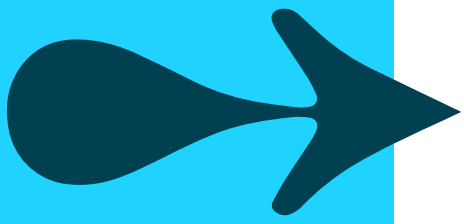


- Operators &, | and ^ perform bitwise operations when the operands are integral numeric types

38

38

BOOLEAN LOGICAL OPERATOR EXAMPLES

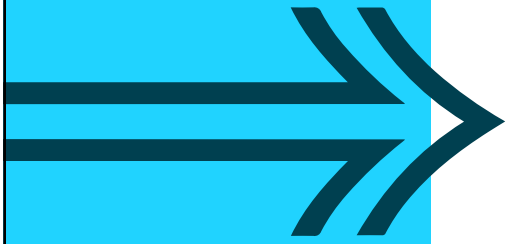


```
// logical negation
bool passed = false;
Console.WriteLine(!passed);  // output: True
Console.WriteLine(!true);    // output: False


// logical exclusive OR
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False

// logical OR
Console.WriteLine(true | true); // output: True
Console.WriteLine(true | false); // output: True
Console.WriteLine(false | true); // output: True
Console.WriteLine(false | false); // output: False

// logical AND
Console.WriteLine(true & true); // output: True
Console.WriteLine(true & false); // output: False
Console.WriteLine(false & true); // output: False
Console.WriteLine(false & false); // output: False
```

39

39

BOOLEAN LOGICAL OPERATOR EXAMPLES: & VERSUS &&



```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

// Logical AND: method SecondOperand is always called
bool a = false & SecondOperand();
Console.WriteLine($"{nameof(a)} is {a}");
// Output:
// Second operand is evaluated.
// a is False

bool b = true & SecondOperand();
Console.WriteLine($"{nameof(b)} is {b}");
// Output:
// Second operand is evaluated.
// b is True

// Conditional Logical AND: if first operand is false, method SecondOperand is not invoked
// This is short-circuit evaluation
bool c = false && SecondOperand();
Console.WriteLine($"{nameof(c)} is {c}");
// Output:
// c is False

bool d = true && SecondOperand();
Console.WriteLine($"{nameof(d)} is {d}");
// Output:
// Second operand is evaluated.
// d is True
```

40

40

BOOLEAN LOGICAL OPERATOR EXAMPLES: | VERSUS ||

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

// Logical OR: method SecondOperand is always called
bool a = true | SecondOperand();
Console.WriteLine($"{nameof(a)} is {a}");
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine($"{nameof(b)} is {b}");
// Output:
// Second operand is evaluated.
// True

// Conditional Logical OR: if first operand is true, method SecondOperand is not invoked
// This is short-circuit evaluation
bool c = true || SecondOperand();
Console.WriteLine($"{nameof(c)} is {c}");
// Output:
// True

bool d = false || SecondOperand();
Console.WriteLine($"{nameof(d)} is {d}");
// Output:
// Second operand is evaluated.
// True
```

41

41

BITWISE AND SHIFT OPERATORS

- C# provides standard bitwise and shift operators:

~	**Bitwise complement**
<<	**Left shift**
>>	**Right shift**
&	**Logical AND**
|	**Logical OR**
^	**Logical exclusive OR**

Bitwise operators operate on the integral types at the binary level:

- **~** reverses all bits
- **<<** shifts the bits *n* places to the left
- **>>** shifts the bits *n* places to the right

42

42

QA

BITWISE AND SHIFT OPERATOR: EXAMPLES

~ << >>

```
// bitwise complement
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011

// left shift
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000

// right shift
uint i = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(i, toBase: 2),4}");

uint j = i >> 2;
Console.WriteLine($"After:  {Convert.ToString(j, toBase: 2),4}");
// Output:
// Before: 1001
// After:    10
```

43

43

QA

BITWISE AND SHIFT OPERATOR: EXAMPLES

& ^ |

```
// logical AND &
uint d = 0b_1111_1000;
uint e = 0b_1001_1101;
uint f = d & e;
Console.WriteLine(Convert.ToString(f, toBase: 2));
// Output:
// 10011000

// logical Exclusive OR ^
uint l = 0b_1111_1000;
uint m = 0b_0001_1100;
uint n = l ^ m;
Console.WriteLine(Convert.ToString(n, toBase: 2));
// Output:
// 11100100

// logical OR |
uint o = 0b_1010_0000;
uint p = 0b_1001_0001;
uint q = o | p;
Console.WriteLine(Convert.ToString(q, toBase: 2));
// Output:
// 10110001
```

44

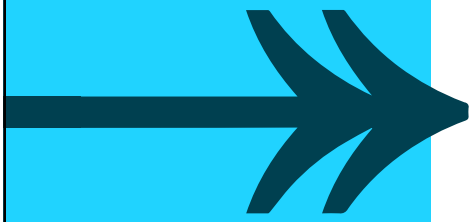
44

EQUALITY OPERATORS

- C# provides two equality operators:

==	**Equality operator**
!=	**Inequality operator**

- The equality operator returns *true* if both operands are equal, *false* otherwise
- Value types are equal if their *values* are equal
- Reference types are equal if they *refer* to the same object

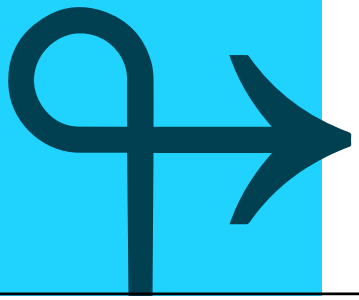


45

45

EQUALITY OPERATOR EXAMPLES: ==

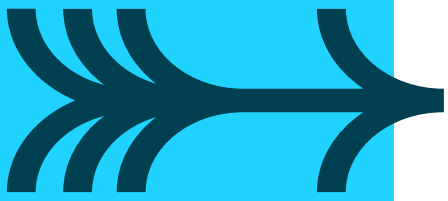
```
// value type equality
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b);  // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2);  // output: False
Console.WriteLine(c1 == char.ToLower(c2));  // output: True


// reference type equality
var car1 = new Car();
var car2 = new Car();
var car3 = car1;
Console.WriteLine(car1 == car2);  // output: False
Console.WriteLine(car1 == car3);  // output: True

// string equality
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower());  // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3);  // output: False
```



46

46

EQUALITY OPERATOR EXAMPLES: !=



```
// value type inequality
int c = 1 + 2 + 3 + 4;
int d = 6;
Console.WriteLine(c != d);  // output: True

// string inequality
string s4 = "Hello";
string s5 = "Hello";
Console.WriteLine(s4 != s5);  // output: False

// reference type inequality
object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2);  // output: True
```

47

47

PARSE

The primitive value types are all able to read in a string and convert to their type using the type's **.Parse()** method.

```
int i          = int.Parse("42");
decimal dec    = decimal.Parse("42.0");
double dbl     = double.Parse("123.45");
```

48

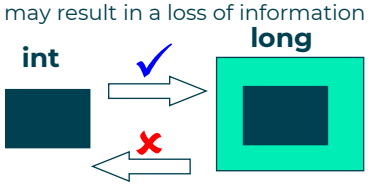
INTEGER ARITHMETIC & CONVERSIONS

Arithmetic works on all numeric types

- Result type is the widest of operand types, but `int` if both narrower
- Cannot mix `decimal` and `float`, or `ulong` and any signed type

You may need to cast the result

- *implicit* casts silently widen to a larger type e.g. `int` to `long`
- *explicit* casting is necessary to do the reverse operation, because it may result in a loss of information



49

INTEGRAL CONVERSIONS EXAMPLE

```
int i = 4;
long l = i; // implicit cast (widening)
l += 3_000_000; // 3 million
//i = l; // implicit cast (narrowing so compile error)
i = (int)l; // explicit cast required to acknowledge potential data loss
Console.WriteLine(i);
// output: 3000004

int j = 4;
long k = j; // implicit cast (widening)
k += 3_000_000_000; // int has a max value of 2_147_483_647
//j = k; // implicit cast (narrowing so compile error)
j = (int)k; // explicit cast required to acknowledge potential data loss
Console.WriteLine(j); // integer overflow results in an incorrect value
// output: -1294967292
```

50

CASTING: REFERENCE TYPE EXAMPLE

Forcing the compiler to convert from one datatype to another:

- This won't compile :

```
static void Main(string[] args) {
    Dog d = GetMammal();
    d.Bark();
}
static Mammal GetMammal() {
    return new Dog();
}
```

I can see that it returns a Dog but the compiler is not allowed to use this information

- It needs a cast to compile

```
Dog d = (Dog)GetMammal();
d.Bark();
```

- Or this

```
((Dog)GetMammal()).Bark();
```

51

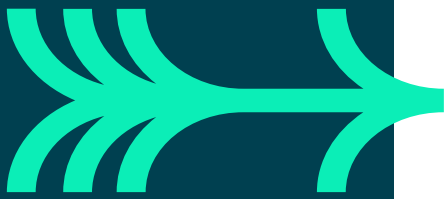
ACTIVITY: Exercise 3

52

52

SUMMARY

- Identifiers
- Comments
- Variables
- Built-in types
- Value types
- Reference types
- Creating variables: Value and reference types
- The var keyword
- Operators
- Parse and casting



53

53