Programming Principles

Semester 1, 2022

Dr Alan Woodley

Tarang Janawalkar

Contents

Contents **1**

1 Programming **5**
1.1 Analysis . . . 5
1.2 Design . . . 5
1.3 Implementation . . . 5
1.4 Testing . . . 5

2 Types and Expressions **6**
2.1 Expressions . . . 6
2.2 Types . . . 6
2.3 Type Conversion . . . 7
2.3.1 Implicit Conversion . . . 8
2.3.2 Explicit Conversion . . . 9
2.4 Operators . . . 10
2.5 Characters . . . 11
2.6 Strings . . . 11
2.6.1 String Indexing . . . 11
2.6.2 Immutability . . . 12
2.6.3 Escape Sequences . . . 12
2.6.4 Verbatim String Literals . . . 12
2.6.5 Format Strings . . . 13
2.6.6 Numeric to String Conversion . . . 13
2.6.7 String to Numeric Conversion . . . 14

3 Structured Programming **14**
3.1 Sequence . . . 14
3.1.1 Blocks . . . 14
3.1.2 Nested Blocks . . . 15
3.2 Selection . . . 15
3.2.1 If Statements . . . 15
3.2.2 If-Else Statements . . . 16
3.2.3 Nested if Statements . . . 16
3.2.4 Cascading if Statements . . . 17
3.2.5 Switch Statements . . . 17
3.3 Iteration . . . 18
3.3.1 While Statements . . . 18
3.3.2 Do-While Statements . . . 18
3.3.3 For Statements . . . 19
3.4 Jump Statements . . . 19
3.5 Boolean Expressions . . . 19
3.5.1 Comparison Operators . . . 19
3.5.2 Logical Operators . . . 20

4 Collections **21**
4.1 Arrays . . . 21
4.2 Access . . . 21
4.3 Array Representation in Memory . . . 22
4.4 Default Values . . . 22
4.5 ForEach Loops . . . 22
4.6 Compound Arrays . . . 23
4.6.1 Parallel Arrays . . . 23
4.6.2 Multidimensional Arrays . . . 23
4.7 Lists . . . 24

5 Methods **25**
5.1 Programming Complexity . . . 25
5.2 Declaring Methods . . . 26
5.2.1 Access Modifiers . . . 26
5.2.2 Return Types . . . 26
5.2.3 Identifiers . . . 26
5.2.4 Parameter List . . . 27
5.2.5 Statements . . . 27
5.3 Invoking Methods . . . 27
5.3.1 Argument List . . . 27
5.3.2 Control Return . . . 28
5.4 Argument Types . . . 28

6 Command Line Arguments **28**
6.1 Visual Studio . . . 28
6.2 Command Line Interfaces . . . 28
6.2.1 Windows DOS . . . 29
6.2.2 UNIX . . . 29
6.3 Working Directory . . . 29
6.3.1 Windows DOS . . . 29
6.3.2 UNIX . . . 29
6.4 Accessing Command Line Arguments . . . 30
6.5 Debugging . . . 30

7 The File System **30**
7.1 Directories . . . 30
7.2 File I/O . . . 31
7.2.1 File Existence . . . 31
7.2.2 Reading from a File . . . 31
7.2.3 Writing to a File . . . 32

8 Classes **32**
8.1 Objects . . . 32
8.2 Access Modifier . . . 32
8.2.1 Public . . . 33
8.2.2 Internal . . . 33

8.2.3 Protected . . . 33
8.2.4 Protected Internal . . . 33
8.2.5 Private Protected . . . 33
8.2.6 Private . . . 33
8.3 Members . . . 33
8.3.1 Static Members . . . 33
8.3.2 Instance Members . . . 34
8.4 Methods . . . 34
8.5 Constructors . . . 34
8.6 Fields . . . 34
8.7 Properties . . . 36
8.7.1 Auto-Implemented Properties . . . 36
8.7.2 Example . . . 36
8.8 Overloading . . . 38
8.9 Class Relationships . . . 41
8.9.1 Composition . . . 41
8.9.2 Aggregation . . . 42
8.9.3 Association . . . 42
8.10 Namespaces . . . 43

9 Object-Oriented Design 43
9.1 Multiple Classes . . . 43
9.2 Modular Decomposition . . . 44
9.3 Functional Decomposition . . . 44
9.4 Abstraction . . . 44
9.5 Encapsulation . . . 44
9.6 User Stories . . . 44
9.7 Software Development Model . . . 45
9.7.1 Waterfall Model . . . 45
9.7.2 Interactive and Incremental Design . . . 45
9.8 Object-Oriented Design . . . 45

10 Unified Modelling Language 46
10.1 Class Diagrams . . . 46
10.2 Classes . . . 46
10.3 Interfaces . . . 46
10.4 Properties and Operations . . . 46
10.4.1 Property . . . 46
10.4.2 Operation . . . 47
10.5 Visibility . . . 47
10.6 Derived '/' . . . 47
10.7 Multiplicity . . . 47
10.8 Property Modifiers . . . 47
10.9 Operation Signatures . . . 48
10.10 Operation Properties . . . 48
10.11 Class Relationships . . . 48

11 Inheritance **48**
11.1 Methods 49
11.2 Virtual Methods 49
11.3 Object Class 50

12 Polymorphism **51**
12.1 Abstract Classes 51
12.2 Static and Dynamic Types 52
12.3 Polymorphic Assignment 53
12.4 Interfaces 53
12.5 Interface Design 54
12.6 Future Expansion 54
12.7 Liskov Substitution Principle 54

1 Programming

Definition 1.1. Programming is the process of designing and building an executable computer program to accomplish a specific computing result or to perform a specific task.

Programming involves:

1. Analysis
2. Design
3. Implementation
4. Testing

1.1 Analysis

Identify:

- the problem
- data, inputs, and outputs
- relationships between inputs and outputs
- constraints

1.2 Design

- Specify modules required to implement the solution
- Determine how modules will integrate with the system and other modules
- Create tests for individual modules
- Create tests for the complete system

1.3 Implementation

- Select suitable algorithms and data structures for design
- Write code to implement algorithms and data structures

1.4 Testing

- Before writing code, create a testing environment to validate the program

2 Types and Expressions

2.1 Expressions

Definition 2.1 (Expressions)**.** An expression is a combination of values, variables, and operators. In interactive mode, an interpreter evaluates expressions and displays the result of this expression. However, in a script, we must first compile the entire program into an executable format, before we can perform any tasks.

Definition 2.2 (Type)**.** The type of an expression is *the kind of data* the expression carries.

Definition 2.3 (Variables)**.** Variables are a kind of expression which have an **identity** and a **value**. The **value** of a variable may change as a program runs, however in a statically typed language, the **type** of each variable is specified before it can be used, and never changes.
Variables can be declared as follows

```
TYPE IDENTIFIER;
TYPE IDENTIFIER = EXPRESSION;
```

In the first instance, we declare the type of the variable without initialisation. In the second instance, we declare and initialise the variable.

Definition 2.4 (Literal)**.** The term *literal* refers to the literal representation of a value. For example, when disambiguating between the variable `dog` and the string `"dog"` we would say the *"variable dog"* vs. the *"string literal dog"*.

C# identifiers must take the following into account

- Identifiers can contain letters, digits and the underscore character (_)
- Identifiers must begin with a letter
- Identifiers cannot contain whitespaces
- Identifiers are case-sensitive ("`Foo`" and "`foo`" are different variables)
- Reserved words such as C# keywords cannot be used as identifiers

2.2 Types

There are 9 integers and 3 floating-point types in C#, each with a different size and range. The minimum and maximum values of any type can be determined using `TYPE.MinValue` and `TYPE.MaxValue`.

C# type	Size	Range
`sbyte`	8-bit	-2^7 to $2^7 - 1$
`byte`	8-bit	0 to $2^8 - 1$
`short`	16-bit	-2^{15} to $2^{15} - 1$
`ushort`	16-bit	0 to $2^{16} - 1$
`int`	32-bit	-2^{31} to $2^{31} - 1$
`uint`	32-bit	0 to $2^{32} - 1$
`long`	64-bit	-2^{63} to $2^{63} - 1$
`ulong`	64-bit	0 to $2^{64} - 1$

Table 1: Integer types in C#.

C# type	Size	Range	Precision
`float`	32-bit	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	6 to 9 digits
`double`	64-bit	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15 to 17 digits
`decimal`	128-bit	$\pm 1.0 \times 10^{-28}$ to 7.9228×10^{28}	28 to 29 digits

Table 2: Floating-point types in C#.

2.3 Type Conversion

By default, C# automatically assigns the `int`, `uint`, `long`, or `ulong` type to any integer depending on the size and sign of the provided number. Any floating-point number is instantiated as a `double`.

```
$ (100).GetType()
[System.Int32]
$ (4294967295).GetType()
[System.UInt32]
$ (-4294967295).GetType()
[System.Int64]
$ (100.0).GetType()
[System.Double]
```

To override this behaviour we can add a suffix to the number.

Type	Suffix
`uint`	`u`
`long`	`l`
`ulong`	`u`, `l` or `ul`
`float`	`f`
`double`	`d`
`decimal`	`m`

Table 3: Type suffixes for numeric types.

If a literal is prefixed with `u`, its type is the first of the following types in which its value can be represented: `uint`, `ulong`. Similarly, if a literal is prefixed with `l`, its type is the first of the following types in which its value can be represented: `long`, `ulong`. If the value of an integer is within the range of the destination type, the value can be implicitly converted to the remaining integer types.

2.3.1 Implicit Conversion

Implicit conversions do not require special syntax as the conversion always succeeds, and no data is lost. The following diagram illustrates all possible implicit conversions for numeric types where the direction of the arrow indicates possible implicit conversions where intermediate types can be skipped. Note that all integer types can be converted to floating-point types.

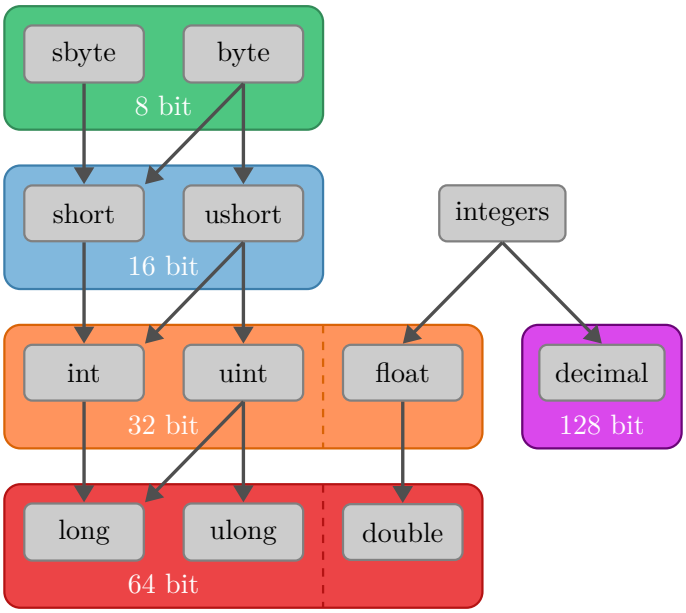

Figure 1: Numeric type implicit conversions in C#.

For example

```
$ // 8-bit unsigned integer to 64-bit signed integer
$ byte b = 32; Console.WriteLine($"{b} {b.GetType()}")
32 System.Byte
$ long l = b; Console.WriteLine($"{l} {l.GetType()}")
32 System.Int64

$ // 16-bit signed integer to double precision floating-point number
$ short s = 30000; Console.WriteLine($"{s} {s.GetType()}")
30000 System.Int16
$ double d = s; Console.WriteLine($"{d} {d.GetType()}")
30000 System.Double
```

2.3.2 Explicit Conversion

When a conversion cannot be made without risking losing information, the compiler requires that we perform an explicit conversion using a **type cast**. The syntax for a type cast is as follows

```
(NEW_TYPE) EXPRESSION
```

For example

```
$ // Decimal to single precision floating-point number
$ decimal pi = 3.14159265358979323m; Console.WriteLine($"{pi} {pi.GetType()}")
3.14159265358979323 System.Decimal
$ float fPi = (float) pi; Console.WriteLine($"{fPi} {fPi.GetType()}")
3.141593 System.Single

$ // 32-bit unsigned integer to 8-bit signed integer
$ uint u = 9876; Console.WriteLine($"{u} {u.GetType()}")
9876 System.UInt32
$ byte b = (byte) u; Console.WriteLine($"{b} {b.GetType()}")
148 System.Byte
```

In the final example, to understand what is happening in the explicit conversion, we must look at the binary representation of the two integers.

```
$ uint u = 9876; Console.WriteLine(Convert.ToString(u, 2).PadLeft(32, '0'))
00000000000000000010011010010100
$ byte b = (byte) u; Console.WriteLine(Convert.ToString(b, 2).PadLeft(8, '0'))
10010100
```

Notice that the value is determined by copying the 8 least significant bits from the 32-bit unsigned integer.

2.4 Operators

The following table lists the C# operators starting with the highest precedence to the lowest.

Operators	Category
`x.y`, `f(x)`, `a[i]`, `x++`, `x--`, `x!`, `x->y` and other keywords	Primary
`+x`, `-x`, `!x`, `~x`, `++x`, `--x`, `^x`, `(T)x`, `await`, `& x`, `*x`, `true`, `false`	Unary
`x..y`	Range
`switch`, `with`	—
`x * y`, `x / y`, `x % y`	Multiplicative
`x + y`, `x - y`	Additive
`x << y`, `x >> y`	Shift
`x < y`, `x > y`, `x <= y`, `x >= y`, `is`, `as`	Relational and type-testing
`x == y`, `x != y`	Equality
`x & y`	Logical `AND`
`x ^ y`	Logical `XOR`
`x | y`	Logical `OR`
`x && y`	Conditional `AND`
`x || y`	Conditional `OR`
`x ?? y`	Null-coalescing operator
`c ? t : f`	Conditional operator
`x = y`, `=>` and shorthand assignments	Assignment and lambda declaration

Table 4: Precedence of various operators in C#.

In C#, arithmetic operations behave as expected.

```
$ 123 + 12
135
$ 123 - 12
111
$ 123 * 12
1476
$ 123 / 12
10
$ 123 % 12
3
```

All results in the example above are of type `System.Int32`. Binary operators always convert the resulting data type to the data type of the argument with the largest size in memory (with a

few exceptions when converting between floating-point types). Hence, division between two 32-bit integers truncates any floating-point precision.

```
$ 123 / 12
10
$ 123.0 / 12
10.25
$ 123 / 12.0
10.25
$ 123.0 / 12.0
10.25
```

Here only the latter three expressions are converted to `System.Double`.

2.5 Characters

A character type represents a **single** Unicode UTF-16 character. Character objects can be implicitly converted to 16-bit unsigned integers, and support the comparison, equality, increment and decrement operators. A character is initialised using single quotation marks (`' '`).

```
$ char c = 'A'; c
'A'
$ c.GetType()
[System.Char]
$ c++
'B'
$ (ushort) c
69
$ c == 69
true
```

2.6 Strings

A string is a sequential read-only collection of character objects that is initialised using double quotation marks (`" "`).

```
$ string s = "Hello, World!"; s
"Hello, World!"
$ s.GetType()
[System.String]
```

2.6.1 String Indexing

The characters in a string can be accessed by position using indexing (starting at 0).

```
$ s[0]
'H'
$ s[s.Length - 1]
'!'
```

2.6.2 Immutability

In C#, string objects are immutable meaning that the string cannot be modified in memory. If a new string is assigned to this object, it will simply point to a new location in memory.

```
$ string s = "String with tyop."; s
"String with tyop."
$ s[14] = 'p';
(1,1): error CS0200: Property or indexer 'string.this[int]' cannot be assigned to
 ↪  -- it is read only
$ s = "String without typo."
"String without typo."
```

2.6.3 Escape Sequences

To use special characters such as newlines, tabs, backslashes, or double quotation marks, we must use an escape sequence.

```
$ string s = "This is a quotation mark \".\nThis line appears on a new line."; s
"This is a quotation mark \".\nThis line appears on a new line."
```

Note that the string is evaluated as a string literal. To view this string verbatim, we must use `Console.WriteLine`.

```
$ Console.WriteLine(s)
This is a quotation mark ".
This line appears on a new line.
```

2.6.4 Verbatim String Literals

If a string contains many escape sequences we can use verbatim strings for convenience.

```
$ string s = @"String with multiple escape sequences ""This is a quote"".
This line appears on a new line.";
$ Console.WriteLine(s)
String with multiple escape sequences "This is a quote".
This line appears on a new line.
```

2.6.5 Format Strings

To dynamically determine a string at runtime, we can use format strings. There are two methods to create format strings: string interpolation and composite formatting. String interpolation allows us to reference variable names directly inside a string. Interpolated strings are identified by the dollar sign.

```
$ int a = 40; int b = 13;
$ $"Given a = {a} and b = {b}, a + b = {a + b}"
"Given a = 40 and b = 13, a + b = 53"
```

Composite formatting uses placeholders for variables which must be provided in order of reference. Here the same variable can be referenced many times in a string.

```
$ int a = 40; int b = 13;
$ string.Format("Given a = {0} and b = {1}, a + b = {2}", a, b, a + b)
"Given a = 40 and b = 13, a + b = 53"
$ string.Format("a + b = {2} where a = {0} and b = {1}", a, b, a + b)
"a + b = 53 where a = 40 and b = 13"
$ string.Format("We can reference `a` twice, here {0} and here {0}", a)
"We can reference `a` twice, here 40 and here 40"
```

2.6.6 Numeric to String Conversion

Strings can be concatenated with numeric variables

```
$ int a = 25;
$ "The temperature is " + a + " degrees."
"The temperature is 25 degrees."
```

As the + operator is evaluated from left to right, the following string concatenation will not evaluate the sum of 1, 2, and 3.

```
$ "sum = " + 1 + 2 + 3
"sum = 123"
```

The `ToString()` method can be accessed from all numeric types, with a format specifier which indicates the number of precision to display.

```
$ (1498).ToString("G3")
"1.5E+03"
$ (1498).ToString("F3")
"1498.000"
$ (1498).ToString("C2")
"$1,498.00"
```

These format specifiers can be applied directly in interpolated strings.

```
$ int i = 1498;
$ $"{i:G3}, {i:F3}, {i:C2}"
"1.5E+03, 1498.000, $1,498.00"
```

We can also add specify padding in interpolated strings.

```
$ decimal pi = 3.14159265358979323m;
$ $"Pi with left padding {pi, 10:F6}"
"Pi with left padding   3.141593"
```

For more information see: *Custom numeric format strings.*

2.6.7 String to Numeric Conversion

We can convert a string to a number by calling the `Parse` method found on numeric types, or by using methods in the `System.Convert` class.

```
$ double.Parse("2.718281")
2.718281
$ Convert.ToDouble("2.718281")
2.718281
```

3 Structured Programming

Structured programming relies on three constructs: sequence, selection and iteration. These help us control the flow of our programs.

3.1 Sequence

3.1.1 Blocks

In C# we can group statements together inside a scope by using braces `{ }`. The statements inside this block are executed in order as a single instruction.

```
$ int i = 5;
. {
.     i = 10;
.     Console.WriteLine(i);
. }
10
$ Console.WriteLine(i);
10
```

Here `i` is accessed as an enclosing locally scoped variable. *This behaviour is akin to blocks defined in selection and iteration structures.*

3.1.2 Nested Blocks

Here is another example that utilises nested blocks and demonstrates access.

```
$ // Global scope
. {
.     // Block 1
.     int i = 5;
.     Console.WriteLine(i);
.     {
.         // Block 2
.         i += 5;
.         Console.WriteLine(i);
.     }
.     // i = 10
. }
$ Console.WriteLine(i);
5
10
(1,19): error CS0103: The name 'i' does not exist in the current context
```

In this example, we see that the variable `i` is local to block 1 and therefore accessible to block 2. However, the converse is not true. `i` cannot be accessed by its enclosing scope as local variables are destroyed when a block ends. We also cannot declare a variable in a block that shares its name with another variable in its enclosing local scope.

```
$ {
.     int i = 5;
.     {
.         int i = 5;
.     }
. }
(4,13): error CS0136: A local or parameter named 'i' cannot be declared in this
 ↪  scope because that name is used in an enclosing local scope to define a local
 ↪  or parameter
```

3.2 Selection

Selection allows us to choose from a range of different options.

3.2.1 If Statements

If statements have the following syntax.

```
// Single statement
if (CONDITION) STATEMENT;

// Multiple statements
if (CONDITION)
{
    STATEMENTS
}
```

In both cases, `CONDITION` is an expression that returns a Boolean value when evaluated. If this value is `true`, the subsequent statement(s) will be executed. Conversely, if the expression yields `false`, the subsequent statement(s) will be ignored and control passes to the next statement after the `if` statement.

3.2.2 If-Else Statements

We can add an alternative statement if `CONDITION` is `false` using an `else` clause.

```
// Single statement
if (CONDITION) STATEMENT_1 else STATEMENT_2;

// Multiple statements
if (CONDITION)
{
    STATEMENTS_1
}
else
{
    STATEMENTS_2
}
```

This structure differs from the previous example as either `STATEMENT_1` or `STATEMENT_2` can be executed. This decision depends on the Boolean value returned by the condition.

3.2.3 Nested if Statements

The blocks in an `if` statement also allow us to nest any number of `if` statements to create a complex flow of control.

```
if (CONDITION_1) if (CONDITION_2) STATEMENT_2 else STATEMENT_1;

// Written using braces
if (CONDITION_1)
{
    if (CONDITION_2)
```

```
    {
        STATEMENT_2
    }
}
else
{
    STATEMENT_1
}
```

Generally, nested `if` statements are difficult to read and should be avoided if possible.

3.2.4 Cascading if Statements

An alternative to nested `if` statements are cascading `if` statements. These statements allow us to provide controlled alternatives to an `if-else` statement if the first condition returns `false`.

```
if (CONDITION_1)
{
    STATEMENTS_1
}
else if (CONDITION_2)
{
    STATEMENTS_2
}
else if (CONDITION_3)
{
    STATEMENTS_3
}
...
else
{
    STATEMENTS_N
}
```

In this structure, any statement $1 < i \leq n$ will be executed if and only if all conditions before i yield `false` and `CONDITION_I` yields `true`. The final statement n after the `else` clause is executed if all preceding conditions return `false`. Note that the `else` clause may be omitted.

3.2.5 Switch Statements

A `switch` statement is an alternative to cascading `if` statements and are another kind of multi-way branch.

```
switch (EXPRESSION)
{
    case CONSTANT_1:
```

```
        STATEMENTS_1
        break;
    case CONSTANT_2:
        STATEMENTS_2
        break;
    ...
    default:
        STATEMENTS_N
        break;
}
```

In this structure, `EXPRESSION` is any numeric or string expression, and `CONSTANT` is a literal of matching type. This means that `STATEMENTS_I` is executed if `EXPRESSION == CONSTANT_I`. The default branch behaves similarly to an `else` clause, and it is executed if none of the preceding cases are satisfied. Each branch must end with one of the following keywords depending on where the switch statement is defined: `break`, `return`, `goto`, `throw`, or `continue`.

3.3 Iteration

Iterative constructs allow us to repeat statements zero, one, or many times, without making multiple copies of the statement.

3.3.1 While Statements

```
while (CONDITION)
{
    STATEMENTS
}
```

Fundamental semantics:

1. Execute `STATEMENTS` if `CONDITION == true`.
2. Go to Step 1.

3.3.2 Do-While Statements

```
do
{
    STATEMENTS
}
while (CONDITION)
```

Fundamental semantics:

1. Execute `STATEMENTS`.
2. Go to Step 1 if `CONDITION == true`.

3.3.3 For Statements

```
for (INIT; CONDITION; UPDATE)
{
    STATEMENTS
}
```

Fundamental semantics:

1. Execute `INIT`.
2. Execute `STATEMENTS` if `CONDITION == true`.
3. Execute `UPDATE`.
4. Go to Step 2.

Generally, a `while` statement is used when the number of iterations is unknown and `for` statements are used otherwise. Both of these structures can execute statements either zero, one or many times. While uncommon, `do-while` statements are used if we want to execute the loop body at least once.

3.4 Jump Statements

The following statements unconditionally transfer control:

- `break` terminates the closest enclosing iteration or switch statement
- `continue` starts a new iteration of the closest enclosing iteration statement

Note that jump statements can be placed anywhere inside the loop body and that any succeeding statements or the `UPDATE` statement (in the `for` structure) will not be executed.

3.5 Boolean Expressions

A Boolean (`bool`) is a type that has two values, `true` and `false`. Boolean expressions are expressions that when evaluated, yield a Boolean value.

3.5.1 Comparison Operators

Comparison operators are a common Boolean expression:

- `x == y`
- `x != y`
- `x < y`
- `x > y`
- `x <= y`
- `x >= y`

3.5.2 Logical Operators

Logical operators are also Boolean expressions:

- `!a` (not)
- `a && b` (and)
- `a || b` (or)

In C#, logical operators use short-circuit evaluation, meaning that if the left expression guarantees the resulting Boolean value, the right expression is not evaluated. For example

```
$ bool a()
. {
.     Console.WriteLine("a was executed.");
.     return false;
. }
$ bool b()
. {
.     Console.WriteLine("b was executed.");
.     return true;
. }
$ a() && b()
a was executed.
False
```

Here we see that the second function `b()` was not executed as the `AND` operator requires both expressions to be true, so regardless of the value of `b()`, the result will be `false`. Similarly, for `OR`:

```
$ bool a()
. {
.     Console.WriteLine("a was executed.");
.     return false;
. }
$ bool b()
. {
.     Console.WriteLine("b was executed.");
.     return true;
. }
$ b()  a()
b was executed.
True
```

Here only one of the two expressions needs to evaluate to `true` for the result to be true. However, if we reversed the order of the expressions:

```
$ bool a()
. {
.     Console.WriteLine("a was executed.");
.     return false;
. }
$ bool b()
. {
.     Console.WriteLine("b was executed.");
.     return true;
. }
$ a()  b()
a was executed.
b was executed.
True
```

we can see that since `a()` does not guarantee the resulting Boolean value, we must also evaluate `b()`. The `NOT` operator simply negates the value of the Boolean expression.

4 Collections

4.1 Arrays

If we want to effectively manage groups of related objects, we can utilise arrays and collections. When defining an array of type `T`, we must append the type with square brackets `[ ]`.

```
TYPE[] IDENTIFIER;
TYPE[] IDENTIFIER = EXPRESSION;
```

As a result, each item must have the same data type. To specify the array size, we can initialise the variable with the `new` expression.

```
TYPE[] IDENTIFIER = new TYPE[SIZE];

// Alternatively
TYPE[] IDENTIFIER;
IDENTIFIER = new TYPE[SIZE];
```

4.2 Access

To access the array, we can use the identifier for that array. Each object within an array can be distinguished by its subscript (or index). To access the object within an array, we place the index inside square brackets `[i]` and append this to the identifier.

```
$ int[] intArray = new int[3];
$ intArray
int[3] { 0, 0, 0 }
$ intArray[0] = 1; intArray[0]
1
$ intArray[1] = 2; intArray[0]
2
$ intArray[2] = 3; intArray[0]
3
$ intArray
int[3] { 1, 2, 3 }
```

Note that array indexing starts at 0. If the value of the elements inside the array are known when the array is initialised, we can set the element values using the following syntax.

```
TYPE[] IDENTIFIER = new TYPE[SIZE] { element1, element2, ..., elementN };
TYPE[] IDENTIFIER = new TYPE[] { element1, element2, ..., elementN };
TYPE[] IDENTIFIER = { element1, element2, ..., elementN };
```

Note that when using the first method, the number of elements N must be equal to the specified `SIZE`.

4.3 Array Representation in Memory

To be able to access an array by index, the elements must be stored contiguously in memory, that is, stored one after the other. This means that each element i is $i \times S_T$ bytes from $i = 0$, where S_T is the size of the array type.

4.4 Default Values

If an array is declared without initialisation, then each element takes one of the following default values based on the type:

Numeric `0`

Character `\u0000` or `null`

Boolean `false`

4.5 ForEach Loops

The `foreach` statement provides a simple, clean way to iterate through the elements of an array, without using the element index.

```
$ int[] array = { 1, 2, 3 };
$ foreach (int i in array)
```

```
. {
.     Console.WriteLine(i);
. }
1
2
3
```

Note that the values `i` are read-only, and hence this structure does not modify the elements of an array. If we wanted to modify these values, can we use either a **`while`** or **`for`** loop. The following example increments each value of an array using a **`for`** loop.

```
$ int[] array = { 1, 2, 3 };
$ for (int i = 0; i < array.Length; i++)
. {
.     array[i]++;
. }
$ array
int[3] { 2, 3, 4 }
```

4.6 Compound Arrays

When we have two or more arrays of values that share the same indices, we can implement compound arrays to create a correspondence between those arrays.

4.6.1 Parallel Arrays

Parallel arrays are useful when the various arrays hold different types of data. In this method, multiple single dimensional arrays are created.

```
$ string[] item = { "Shoes", "Shirt", "Pants" };
$ int[] price = { 100, 40, 80 };
$ for (int i = 0; i < item.Length; i++)
. {
.     Console.WriteLine($"Item: {item[i]}\tPrice: {price[i]:C}");
. }
Item: Shoes     Price: $100.00
Item: Shirt     Price: $40.00
Item: Pants     Price: $80.00
```

Although there are benefits to using this approach, larger data sets with multiple columns may be better represented in a multidimensional array.

4.6.2 Multidimensional Arrays

Multidimensional arrays behave similarly to regular arrays but use different declaration and access syntax.

```
// 2D arrays
TYPE[,] IDENTIFIER = new TYPE[SIZEX, SIZEY];

// nD arrays
TYPE[,,...] IDENTIFIER = new TYPE[SIZEX, SIZEY, ...];
```

When declaring the type, it is important to specify the correct number of commas (,). The number of commas is equal to the number of dimensions minus 1. When instantiating the array on the right-hand side of the assignment, the length of each dimension is specified as a comma-separated integer. Similar to single-dimensional arrays, we can set the values when we declare the array.

```
$ int[,] array2d = {{ 2, 3 },
.                     { 1, 5 }};
$ array2d
int[2, 2] { { 2, 3 }, { 1, 5 } }
```

Multidimensional arrays are commonly separated at each dimension so that they are easier to read. When accessing the value from a multidimensional array, so that they are easier to read. When accessing the value from a multidimensional array, we must separate each index using a comma.

```
$ int ROW = array2d.GetLength(0);
$ int COL = array2d.GetLength(1);

$ for (int i = 0; i < ROW; i++)
. {
.     for (int j = 0; j < COL; j++)
.     {
.         Console.WriteLine($"The value at ({i}, {j}) is {array2d[i, j]}");
.     }
. }
The value at (0, 0) is 2
The value at (0, 1) is 3
The value at (1, 0) is 1
The value at (1, 1) is 5
```

Here the `GetLength()` methods are used to find the length of both the first and second dimensions.

4.7 Lists

Lists allow us to have variable size arrays, meaning we can add and remove elements during execution.

```
List<TYPE> IDENTIFIER = new List<TYPE>();
List<TYPE> IDENTIFIER = new List<TYPE>{ elements };
```

The first syntax creates a list with 0 elements. We can add to this list using the `Add()` method, and remove from it using the `Remove()` method.

```
$ List<int> list = new List<int>();
$ list
List<int>(0) { }
$ list.Add(3)
$ list.Add(1)
$ list.Add(4)
$ list.Add(5)
$ list.Add(1)
$ list.Add(5)
$ list
List<int>(6) { 3, 1, 4, 5, 1, 5 }
```

By using the `Remove()` method, we can remove the accidental 5 that appears at index 3.

```
$ list.Remove(5)
true
$ list
List<int>(5) { 3, 1, 4, 1, 5 }
$ list.Remove(7)
false
$ list
List<int>(5) { 3, 1, 4, 1, 5 }
```

From the first `Remove()`, only the first `5` was removed from the list. The method also returned a Boolean value indicating that the value `5` existed in the list. When we try to remove the number `7`, the method returns `false`, as the list does not contain a `7`, and hence the list remains unchanged.

5 Methods

5.1 Programming Complexity

Programming logic is usually complex as it involves modelling rules and constraints from the real world. When dealing with complexity, we must guarantee:

- Correctness
  - The software is trustworthy
  - The software meets requirements
  - Valid inputs always produce the required outcomes
- Robustness
  - The program does not crash

- Efficiency
  - The program completes its work quickly
  - The program utilises efficient algorithms
  - The program does not introduce unnecessary loops or convoluted logic
  - The program uses existing standard libraries

We can use methods to address this complexity by decomposing large tasks into smaller tasks.

- Each sub-task handles part of the larger problem
- Each sub-task can be implemented separately as a method or group of methods
- Tests can be developed to target each method individually

As long as each sub-task is correct, robust, and efficient, the overall program will be also. As sub-tasks are simpler than the original problem, these goals are easier to achieve. Methods can also help eliminate duplicate code, so that code is more compact, and by moving them into libraries, we can use them in other projects making programming more productive.

5.2 Declaring Methods

A method has the following syntax

```
[ACCESS_MODIFIER] RETURN_TYPE METHOD_IDENTIFIER(PARAMETER_LIST)
{
    STATEMENTS
}
```

5.2.1 Access Modifiers

Access modifiers will be discussed in a later section.

5.2.2 Return Types

A return type specifies what type of value is returned to the method caller. If a method alters the state of the program and does not necessarily require an output, is a side effect method and its return type is `void`.

5.2.3 Identifiers

The identifier is the name of the method, which can be any valid C# identifier. By convention, method identifiers are camel-case with an upper-case first letter.

5.2.4 Parameter List

The parameter list declares the formal parameters of the method, i.e., what arguments may be passed to the method. The parameter list is a comma-separated list of parameters that have the following syntax

```
[PARAMETER_MODIFIER] TYPE PARAMETER_IDENTIFIER [= DEFAULT_VALUE]
```

The parameter modifier tells the compiler whether to send an argument by reference or by value. There are 4 modifiers to choose from

- `in` — Arguments are passed by reference but cannot be modified.
- `ref` — Arguments are passed by reference and can be modified.
- `out` — Arguments are passed by reference but must be initialised inside the method.
- `params` — A variable number of arguments are passed by value.

By default, if a modifier is not specified, an argument is passed by value. The parameter type specifies what type of argument is to be passed, and it gives the argument an alias within the method's scope. Default values can be assigned to parameters if they are optional. Note that parameters do not need to be re-declared inside the method.

5.2.5 Statements

The statements inside a method can consist of sequences of statements including declarations, expressions, and structured blocks. As a method introduces a nested block, all variables declared in a method body are local to the method.

5.3 Invoking Methods

To call a method we can invoke it using the following syntax

```
METHOD_IDENTIFIER(ARGUMENT_LIST)
```

If a method is defined in another class, we must qualify the name with the appropriate class:

```
CLASS_NAME.METHOD_IDENTIFIER(ARGUMENT_LIST)
```

5.3.1 Argument List

The argument list can be empty or a list of comma-separated expressions, corresponding to the formal parameter list in the same order encountered.

```
[PARAMETER_MODIFIER] EXPRESSION
```

If the corresponding formal parameter is `ref` or `out`, the argument must also be qualified with the corresponding modifier.
If we do not want to provide arguments in order, or if certain optional parameters wish to be skipped, we can do so by providing the name of the parameter identifier using the following syntax.

```
PARAMETER_IDENTIFIER: [PARAMETER_MODIFIER] EXPRESSION
```

This allows us to provide arguments in an arbitrary order.

5.3.2 Control Return

When control is returned to the method caller, the method may provide an output if its return type is non-void. In this event, the method call can be treated as an expression, and it can be assigned to a variable.

```
RETURN_TYPE IDENTIFIER = METHOD_IDENTIFIER(ARGUMENT_LIST)
```

Here the identifier being assigned the output must have the same type as the return type of the method.

5.4 Argument Types

While we can use parameter modifiers to force value type variables to be sent by reference, certain structure types are implicitly passed by reference.
Objects such as arrays are a pointer to a location in memory so that instead of passing a copy of that array into the method, the address of its location is passed.
This allows the array elements to be modified from within the method.

6 Command Line Arguments

When writing a program, we use the `Main` method to indicate the entry point of our program.
This method also allows us to pass arguments that can be specified when the program is run. These arguments are called command line arguments.

6.1 Visual Studio

In Visual Studio, we can supply command line arguments through the Launch profiles UI, which can be accessed through Project » Properties » Debug.

6.2 Command Line Interfaces

A command line interface processes commands using lines of text; ensure that all commands are entered correctly before proceeding.
If we wish to use a command line interface, such as the Windows DOS Command Prompt or Powershell, or the UNIX terminals provided on macOS and Linux distributions, we must ensure that we are working in the correct directory.

6.2.1 Windows DOS

On Windows, launching a terminal will take us to either the user directory,

```
C:\Users\username>
```

or the system directory,

```
C:\Windows\System32>
```

if we launch the terminal as an Administrator.

6.2.2 UNIX

In UNIX operating systems, launching a terminal takes us to the home directory,

```
username@hostname ~ %
```

Note the prompt may look different on Linux distributions, and the % may be replaced by a $ if the user is a superuser (`sudo`).

6.3 Working Directory

We can use the `cd` (change directory) command to set the current directory to the location of our C# project.

6.3.1 Windows DOS

```
C:\Users\username>cd C:\path\to\project
C:\path\to\project>
```

6.3.2 UNIX

```
username@hostname ~ % cd /path/to/project
username@hostname project %
```

After completing these steps, we can execute our program with command line arguments.

```
$ dotnet run <args>
```

6.4 Accessing Command Line Arguments

The `Main` method's signature:

```
static void Main(string[] args)
```

converts all command line arguments to strings, where arguments are separated by spaces. If we wish to maintain the spaces in a string, we can surround that argument with double quotation marks (").

```
$ dotnet run "First argument" Second Third "Fourth argument"
```

The arguments can be accessed with the parameter identifier (`args`) using iteration.

```
$ foreach(string arg in args) Console.WriteLine(arg);
First argument
Second
Third
Fourth Argument
```

6.5 Debugging

During a debugging session, we can watch the variables that are currently on the stack. In Visual Studio we can use the Immediate Window found through Debug ≫ Windows ≫ Immediate Window.
To pause the program during execution, we must enter step mode by placing a breakpoint anywhere in the program's control. This will pause the program before the breakpoint and allow us to view and modify values currently on the stack. This debugging window is very similar to the C# interactive CLI.

7 The File System

7.1 Directories

Using the `System.IO` class, we can view information about the files and directories in the current working directory. We can do so by using the following `using` preprocessor directive:

```
using System.IO;
```

To retrieve information about the project directory we must first fetch the environment directory.

```
string pwd = Environment.CurrentDirectory;
```

We can then use the `DirectoryInfo` class to access metadata about a directory,

```
DirectoryInfo directoryInfo = new DirectoryInfo("PATH\TO\DIRECTORY");
```

and the `FileInfo` class to access metadata about a file.

```
FileInfo fileInfo = new FileInfo("PATH\TO\FILE");
```

This allows us to utilise the various DirectoryInfo and FileInfo methods.

7.2 File I/O

To access the contents of files, we use the `File` type in order to query, read, write, create, copy, or delete a file from a directory.

7.2.1 File Existence

Before we use any IO methods, we must ensure that the file exists. We can use the `Exists` method to accomplish this:

```
File.Exists("PATH\TO\FILE")
```

7.2.2 Reading from a File

Once we have confirmed the file exists, we can read from it using the `StreamReader` class.

```
{
    using StreamReader reader = new StreamReader("PATH\TO\FILE");

    while (!reader.EndOfStream) {
        Console.WriteLine(reader.ReadLine());
    }
}

// Alternatively
using (StreamReader reader = new StreamReader("PATH\TO\FILE"))
{
    while (!reader.EndOfStream)
    {
        Console.WriteLine(reader.ReadLine());
    }
}
```

Note the usage of the `using` statement, this ensures the file is properly closed at the end of the enclosing block.

7.2.3 Writing to a File

Similar to the previous section, we can use the `StreamWriter` class to write to a new file.

```
{
    using StreamWriter writer = new StreamWriter("PATH\TO\FILE");

    writer.WriteLine(CONTENT_TO_ADD);
}

// Alternatively
using (using StreamWriter writer = new StreamWriter("PATH\TO\FILE"))
{
    writer.WriteLine(CONTENT_TO_ADD);
}
```

8 Classes

Classes are used to define a new type of object. They provide a blueprint for creating objects, providing initial values for maintaining state, and implementations of behaviour. To create a class in C#, we can use the following syntax.

```
[ACCESS_MODIFIER] class CLASS_IDENTIFIER
{
    // Class members
}
```

8.1 Objects

If we wish to implement this class we can *instantiate* it, or create an instance of it, using the **new** keyword.

```
CLASS_IDENTIFIER IDENTIFIER = new CLASS_IDENTIFIER();
```

8.2 Access Modifier

All types and type members have an accessibility level. This level controls whether they can be used from other code inside the assembly or other assemblies. An assembly is a `.dll` or `.exe` created by compiling one or more `.cs` file in a single compilation. Additional information is available at Access Modifiers (C# Programming Guide). In order of highest access to lowest:

1. Public
2. Internal

3. Protected
4. Protected internal
5. Private protected
6. Private

8.2.1 Public

Accessible by code in the same assembly or another assembly. The accessibility level of public members of a type is controlled by the accessibility level of the type itself.

8.2.2 Internal

Accessible by code in the same assembly, but not from another assembly.

8.2.3 Protected

Accessible by code in the same class, or in a derived class.

8.2.4 Protected Internal

Accessible by code in the same assembly, or from a derived class in another assembly.

8.2.5 Private Protected

Accessible by code in a derived class in the same assembly.

8.2.6 Private

Accessible by code in the same class.

8.3 Members

A class contains many kinds of members that represent the state of that class through its data and behaviour. Fields, properties, methods, and constructors are kinds of members. These members are placed in the scope of the class they belong to.

8.3.1 Static Members

The static modifier can be used to declare a member which belongs to the type itself, rather than to a specific object. A class can be static so that all its members are also static. This class cannot be instantiated, so its members must be referenced from the class.

```
static class Class
{
    public static void Method() { }
}
```

```
class Program
{
    public static void Main()
    {
        Class.Method();
    }
}
```

Static fields represent the global state across the class. They should be used inside classes only if they are associated with that class. Methods can also be static when they do not depend on any non-static fields.

8.3.2 Instance Members

Instance members are non-static members that belong to each instance of a class and can access the non-static fields for each object. On the other hand, instance fields are unique to each object.

8.4 Methods

As mentioned in Section 5, methods are used to create reusable code. In a class, non-static methods have access to the state of an object, meaning they can be unique to every instance.

8.5 Constructors

Constructors are special methods that are called when an object is created. The constructor may be omitted or used to initialise the fields of an object. A constructor must use the same identifier as the enclosing class and does not have a return type.

```
[ACCESS_MODIFIER] CLASS_IDENTIFIER(PARAMETER_LIST) {
    // Constructor body
}
```

Arguments are passed to the parameter list when the object is instantiated:

```
CLASS_IDENTIFIER IDENTIFIER = new CLASS_IDENTIFIER(ARGUMENT_LIST);
```

The constructor body can then assign parameters to their corresponding fields (or properties). Additionally, the constructor can perform initialisation procedures that manage the state of the class.

8.6 Fields

A field is a variable of any type that is declared inside a class. A class may contain both instance and static fields. The syntax is as follows,

```
[ACCESS_MODIFIER] TYPE IDENTIFIER [= DEFAULT_VALUE];
```

Constant fields use the `const` keyword so that their values are determined at compilation time, and cannot be changed by the program. Constant fields can be either instance or static fields. The following example assigns a name to every instance of the `Person` class while keeping track of how many instances have been made using an incrementor in the constructor.

```
class Person
{
    public static int Count = 0;
    public string Name;

    public Person(string name)
    {
        Name = name;
        Count++;
    }
}

class Program
{
    public static void Main()
    {
        Person Alice = new Person("Alice");
        Console.WriteLine(Person.Count);

        Person Bob = new Person("Bob");
        Console.WriteLine(Person.Count);
    }
}
```

Output:

```
1
2
```

Note that we can use the `this` keyword to distinguish the parameter and field names, if, for example, the parameter was also capitalised:

```
public Person(string Name)
{
    this.Name = Name;
    Count++;
}
```

8.7 Properties

A property is a public member that provides a flexible mechanism to read, write, or compute the value of its associated private field. Properties are special methods called accessors that are commonly used to validate inputs or update other fields that depend on the field being modified. This allows for complex data structures while ensuring the implementation is safe and the data is easily accessible. A property has two accessors; `get` and `set`. The get property accessor is used to return the property value, and the set property accessor is used to assign a new value. Properties start with a capital letter and have the same name as their associated field.

```
private TYPE FIELD_IDENTIFIER;
public TYPE PROPERTY_IDENTIFIER
{
    get
    {
        return FIELD_IDENTIFIER;
    }
    set
    {
        FIELD_IDENTIFIER = value;
    }
}
```

In the example above, the getter simply returns the field associated with the property. Within the setter, the parameter `value` represents the value that is being assigned to the field. We can implement whatever validation we want in this block before the new value is assigned to the field identifier.

8.7.1 Auto-Implemented Properties

If both getter and setter accessors do not require a body, i.e., `get` simply returns the field, and `set` only assigns the incoming value to the field, we can use an auto-implemented property.

```
public TYPE PROPERTY_IDENTIFIER { [ACCESS_MODIFIER] get; [ACCESS_MODIFIER] set; }
```

Here we do not need an additional private field, but rather we can use the `private` access modifier before the `set` keyword. This allows us to quickly create a variable that is either public, read-only, or write-only.

8.7.2 Example

The following example illustrates all of these concepts.

```
class NumberExponent
{
    public int Result { get; private set; }
```

```
    private int exponent;
    public int Exponent
    {
        get
        {
            return exponent;
        }
        set
        {
            if (value > 0)
            {
                exponent = value;
                Result = (int)Math.Pow(number, exponent);
            }
            else
            {
                Console.WriteLine("Exponent cannot be negative.");
            }
        }
    }

    private int number;
    public int Number
    {
        get
        {
            return number;
        }
        set
        {
            number = value;
            Result = (int)Math.Pow(number, exponent);
        }
    }

    public NumberExponent(int number, int exponent)
    {
        // Set values using property setters
        Number = number;
        Exponent = exponent;
    }
}

class Program
{
```

```
    public static void  Main()
    {
        NumberExponent n = new NumberExponent(2, 2);
        Console.WriteLine(n.Result);

        n.Number = 3;
        Console.WriteLine(n.Result);

        n.Exponent = -1;
        Console.WriteLine(n.Result);
    }
}
```

Output:

```
4
9
Exponent cannot be negative.
9
```

8.8 Overloading

Method overloading is a feature that allows methods to be declared in multiple ways by altering either the number or type of parameters, along with the return type of the overloaded method. The following example illustrates how a class constructor can be overloaded to accept various sets of arguments.

```
class Person
{
    public string Name;
    public string Email;
    public int Age;

    public Person(string name, string email, int age)
    {
        Name = name;
        Email = email;
        Age = age;
    }

    public Person(string name, string email)
    {
        Name = name;
        Email = email;
        Age = 0;
```

```
    }

    public Person(string name, int age)
    {
        Name = name;
        Email = "";
        Age = age;
    }

    public Person(string name)
    {
        Name = name;
        Email = "";
        Age = 0;
    }
}

static class Program
{
    public static void Main()
    {
        Person a = new Person("A", "A@email.com", 5);
        Person b = new Person("B", "B@email.com");
        Person c = new Person("C", 9);
        Person d = new Person("D");
    }
}
```

If the method signature allows, we can avoid code duplication by passing the parameters of the overloaded methods into another method inside the same class using the `this` keyword, while providing default values for the remaining unknown parameters.

```
class Person
{
    public string Name;
    public string Email;
    public int Age;

    public Person(string name, string email, int age)
    {
        Name = name;
        Email = email;
        Age = age;
    }

    public Person(string name, string email) : this(name, email, 0) { }
```

```
    public Person(string name, int age) : this(name, "", age) { }

    public Person(string name) : this(name, "", 0) { }
}
```

As the `this` keyword is restricted to constructor overloads, we can use an alternative approach in which the overloaded method calls the base method in its body. Here we define a new method called `Information` that prints the fields of the person specified.

```
class Person
{
    ...

    public void Information(Person p)
    {
        Console.WriteLine($"Name: {p.Name} Email: {p.Email} Age: {p.Age}");
    }

    public void Information() { Information(this); }
}

static class Program
{
    public static void Main()
    {
        ...

        a.Information(b);
        a.Information();
    }
}
```

Output:

```
Name: B Email: B@email.com Age: 0
Name: A Email: A@email.com Age: 5
```

Here the `this` keyword is used as a reference to the object that calls the method, rather than the object itself. It is used to pass the caller as an argument to the base method that implements the functionality. In addition to omitting parameters, the set of parameters can be modified provided the set of parameter types is unique. The following example shows valid overloads for a numeric addition method.

```
public int Add(int a, int b); // Base method
public int Add(double a, int b); // First parameter modified
```

```
public int Add(int a, double b); // Second parameter modified
public double Add(double a, double b); // Parameters and return type modified
public int Add(int a, int b, int c); // Additional parameter
```

The following examples are not valid overloads (assuming the above overloads are also implemented).

```
public int Add(int b, int a); // Parameter types are not unique
public double Add(int a, int b); // Parameter types are not unique
```

This is because parameter identifiers cannot be used to distinguish two overloaded methods, hence the program does not know which method to call when invoked with two 32-bit integer arguments.

8.9 Class Relationships

To understand the relationships between classes, we can use the following terminology to describe specific characteristics of a relationship.

8.9.1 Composition

A composition is a "part-of" relationship where class A maintains a reference of class B and manages its lifespan.

```
class B
{
    ...
}
class A
{
    B b = new B();
}
```

Example:

```
class Engine
{
    string manufacturer;
    int horsepower;
    ...
}
class Car
{
    Engine engine = new Engine("Ferrari", 800)
}
```

8.9.2 Aggregation

An aggregation is a "has-a" relationship where class A maintains a reference of class B but does not manage its lifespan. Class A has ownership over class B.

```
class B
{
    ...
}
class A
{
    B b;

    A(B b)
    {
        this.b = b;
    }
}
```

Example:

```
class Address
{
    ...
}
class Person
{
    private Address address;
    public Person(Address address)
    {
        this.address = address;
    }
}
```

8.9.3 Association

An association is a "uses" relationship where class A does not maintain a reference of class B nor manage its lifespan. Both classes are independent of each other.

```
class B
{
    Foo(A a)
    {
        // Do foo
    }
```

```
}
class A
{
    DoFoo(B b)
    {
        b.Foo(this);
    }
}
```

Example:

```
class Course
{
    string name;
    int id;

    void Enrol(Student student)
    {
        // Enrol student
    }
}

class Student
{
    public void Enrol(Course course)
    {
        course.Enrol(this);
    }
}
```

8.10 Namespaces

Namespaces are used to declare scopes containing a set of related objects. We can also nest namespaces using a nested directory structure for large projects. Multiple files can declare the same namespace with different classes to separate functionality.

9 Object-Oriented Design

9.1 Multiple Classes

As the complexity of a program increases, we can create additional classes to maintain the structure of the program. To do so, we can either create a class in a new file under the same namespace; or in the same file by declaring a new class scope.

9.2 Modular Decomposition

Modular decomposition partitions code into a collection of modules. The quality of a modular decomposition is assessed based on its coupling and cohesion.

- Cohesion — Code within a module is closely related
- Coupling — The interaction between code from different modules is minimal

In an object-oriented model, the methods required to validate fields or perform other unassociated tasks should limit access to other classes. Access to relevant fields and methods from other classes should be limited to avoid unnecessary complexity.

9.3 Functional Decomposition

Prior to object-oriented design, functionality was divided into several smaller functions. Modules were then derived by grouping related functions. However, as a program evolves, existing modules will need to be adapted as new functionality is added, or if existing functionality is modified or removed. This results in a time penalty as users need to revisit old codebases, in order to reuse them for future projects.

9.4 Abstraction

Abstraction is a concept that generalises the underlying complexity and implementation details of an object.

9.5 Encapsulation

Encapsulation restricts the access to logically related data allowing us to hide the internal functionality acting on logically related data inside functions.

9.6 User Stories

The requirements of a task are often expressed as a set of "user stories". These stories are a simple description of required functionality and are told from the perspective of a particular type of user. They consist of:

- the type of user — a particular persona
- action — what they want to achieve
- benefit/value — why its important

User stories keep the focus on the use, enable collaboration, drive creative solutions, and create momentum. When writing user stories, consider:

- the definition of "done"
- outlining subtasks or tasks
- user personas

- ordered steps
- feedback
- time

Some limitations to user stories are that they scale up very quickly, they can sometimes be vague, informal or incomplete, they can lack non-functional requirements, and also don't represent how to fulfil a task.

9.7 Software Development Model

The following sections discuss various software development models.

9.7.1 Waterfall Model

The waterfall model can be described by the following steps:

1. Analysing requirements
2. Design
3. Implementation
4. Testing
5. Deployment

This model provides clearly defined stages with well understood milestones, and it is easy to manage due to the rigidity of the model. Phases are processed and completed one at a time, and results are well documented. This model is well suited for smaller projects where requirements are well understood. Some disadvantages to this are that no working software is produced until the final stages of the development lifecycle. This leads to large amounts of risk and uncertainty for complex projects as the project cannot accommodate changing requirements.

9.7.2 Interactive and Incremental Design

The interactive and incremental model focuses on smaller subtasks which are completed one by one so that the process as a whole is iterative. This makes handling complex problems easier and allows for changes to the requirements through refactoring. Refactoring is the process of modifying existing code to facilitate newly required functionality. It is important to refactor code before adding additional functionality.

9.8 Object-Oriented Design

In the initial stages of a project, candidate classes should be identified such that they are tangible entities and abstract concepts. A technique for identifying classes is by looking for nouns. When designing methods, we need to decide which class a method belongs to based on what the class is responsible for. Methods are found by looking for verbs in a project outline. During this stage, we must also consider the data associated with the program and which class has ownership over the data, so that methods acting on data can be placed in the appropriate class to maintain privacy.

10 Unified Modelling Language

The Unified Modelling Language (UML) is a modelling language that provides a standard for visualising the design of a system.

10.1 Class Diagrams

Class diagrams are a type of UML diagram that describe the structure of a program by showing the program's classes, their attributes, operations, and relationships with other objects. A class with implementation level details is represented as a table with three rows:

Object name
Fields
Methods

10.2 Classes

The name of a class is placed in the first row and is styled **bold**. The following styles are also applied where applicable:

- *Italics* — Abstract classes
- Underline — Static classes

Class

10.3 Interfaces

Interfaces have the same rules as classes, but are preceded by "«Interface»":

«Interface» **Interface**

10.4 Properties and Operations

For the following sections, brackets ([]) are used to denote optional fields. Properties (or fields) and operations (or methods) are represented using the following syntax:

10.4.1 Property

```
[visibility] ['/'] property-name [':' property-type] ['['multiplicity']']
    ['=' |default|-value] [property-modifiers]
```

10.4.2 Operation

```
[visibility] signature [operation-properties]
```

The following sections explain each of the placeholders.

10.5 Visibility

Visibility refers to an objects access level, and is represented by the following symbols:

+ Public

∼ Internal

Protected

− Private

10.6 Derived '/'

The forward-slash (/) is used to represent a derived field, meaning its value depends on the value of other fields in that class.

10.7 Multiplicity

Multiplicity allows us to specify cardinality, i.e., the number of elements in a collection. Multiplicity is specified using a range of values, and it uses the following syntax:

```
a..b
```

which means a collection can contain n elements such that $a \leq n \leq b$. For unbounded ranges, we can use an asterisk ($*$), i.e., "1 or more" would have the multiplicity: `1..*`.

10.8 Property Modifiers

A property modifier can be one of the following:

Modifier	Description
id	The property is an identifier
readOnly	The property is read-only
ordered	The property is ordered
unique	The property is unique
nonunique	The property may have duplicate values
sequence	The property is an ordered bag (unique = false and ordered = true)
union	The property is a derived union of its subsets
redefines P	The property redefines an inherited property named P
subsets P	The property is a subset of the property named P
<property-constraint>	A constraint that applies to the property (i.e., $P > 0$)

These property modifiers are placed between braces {} and separated by commas.

10.9 Operation Signatures

An operation's signature takes the following syntax:

```
operation-name'('[parameter-list]')' [':' return-type ['['multiplicity']']]
```

The parameter list has similar syntax to C#, but identifiers and types are swapped.

10.10 Operation Properties

An operation property can be one of the following:

Modifier	Description
redefines O	The operation redefines an inherited operation named O
query	The operation does not change the state of the system
ordered	The values of the return parameter are ordered
unique	The values of the return parameter are unique
<operation-constraint>	A constraint that applies to the operation (i.e., $O > 0$)

These property modifiers are placed between braces {} and separated by commas.

10.11 Class Relationships

The following arrows are used to show relationships between classes:

Relationship	Arrow	Description
Association	C_1 —— C_2	C_1 uses C_2
Aggregation	C_1 ◇—— C_2	C_1 has a C_2
Composition	C_1 ——◆ C_2	C_1 is part of C_2
Generalisation (inheritance)	C_1 ——▷ C_2	C_1 extends C_2
Realisation (implementation)	C_1 - - - - - -▷ C_2	C_1 realises C_2
Dependency	C_1 - - - - - -> C_2	C_1 depends on C_2

11 Inheritance

Inheritance is an object-oriented principle that allows us to create new classes that reuse, extend, and modify the behaviour defined in another class. The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class. For example:

```
class A { }
class B : A { }
```

In this example, class B inherits class A, so it is the derived class, and A is the base class. In a UML class diagram, this is referred to as the generalisation of a derived class.

11.1 Methods

If a method is implemented in a base class, it can be accessed by a derived object, even if the derived class does not implement the method. For example, B has access to the `PrintHello` method, although it does not define it.

```
class A
{
    public void PrintHello() { Console.WriteLine("Hello"); }
}
class B : A { }
...
static void Main()
{
    B b = new B();
    b.PrintHello(); // prints "Hello"
}
```

11.2 Virtual Methods

A virtual method provides a default implementation for a method that may be reimplemented by derived classes. The method can be reimplemented using the `override` keyword. If a derived class does not override a virtual method, the base class implementation is used. For example, B reimplements the `PrintHello` method, while C uses A's default implementation.

```
class A
{
    public virtual void PrintHello() { Console.WriteLine("Hello"); }
}
class B : A
{
    public override void PrintHello() { Console.WriteLine("Hello B"); }
}
class C : A { }
...
static void Main()
{
    B b = new B();
```

```
    C c = new C();
    b.PrintHello(); // prints "Hello B"
    c.PrintHello(); // prints "Hello"
}
```

Note that the signature and modifiers must match the base class implementation. Although B reimplements this method, it can still call the base class's implementation using the `base` keyword.

```
class A
{
    public virtual void PrintHello() { Console.WriteLine("Hello"); }
}
class B : A
{
    public override void PrintHello()
    {
        base.PrintHello();
        // Perform additional tasks
    }
}

static void Main()
{
    B b = new B();
    b.PrintHello(); // prints "Hello B"
}
```

This technique should be utilised if a reimplementation only *adds* to a base classes methods rather than modifying their implementations. This avoids repeating code.

11.3 Object Class

In C# all classes are implicitly derived from the `Object` class. This class contains built-in methods such as `Equals()`, `GetHashCode()`, and `ToString()`. These are all virtual methods that can be reimplemented in user defined classes.

Equals Determines whether the specified object is equal to the current object.

GetHashCode Serves as the default hash function.

ToString Returns a string that represents the current object.

The following example overrides the `Equals` and `ToString` methods.

```
class Point
{
    public int X, Y;
```

```
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }
    public override bool Equals(Object obj)
    {
        // Check for nullity and compare types
        if ((obj == null) || ! this.GetType().Equals(obj.GetType()))
        {
            return false;
        }
        else {
            // Type cast obj onto Point
            Point p = (Point) obj;
            return (x == p.X) && (y == p.Y);
        }
    }
    public override string ToString() { return $"({X}, {Y})"; }
}
...
static void Main()
{
    Point a = new Point(2, 3);
    Point b = new Point(2, 3);
    Point c = new Point(4, 5);

    a.Equals(b); // true
    b.Equals(c); // false
    c.ToString() // prints "(4, 5)"
}
```

12 Polymorphism

Polymorphism allows us to declare a common operation that is implemented differently across derived types.

12.1 Abstract Classes

Abstract classes allow us to develop partially implemented classes without the need to fully implement methods like in concrete classes. This allows us to declare abstract methods that have no implementation in the base class, but must be implemented (using `override`) in all derived classes. It follows that abstract classes cannot be instantiated. To declare an abstract class or method, we can use the `abstract` keyword. Note that abstract methods can only be declared inside abstract

classes.

```
abstract class Animal
{
    public string Name;

    Animal(string name)
    {
        Name = name;
    }
    public abstract void Speak();
}
class Dog : Animal
{
    Dog(string name) : base(name) { }
    public override void Speak()
    {
        Console.WriteLine("Woof");
    }
}
class Cat : Animal
{
    Cat(string name) : base(name) { }
    public override void Speak()
    {
        Console.WriteLine("Meow");
    }
}
...
static void Main()
{
    Animal dog = new Dog("Jack");
    Animal cat = new Cat("James");

    dog.Speak(); // prints "Woof"
    cat.Speak(); // prints "Meow"
}
```

12.2 Static and Dynamic Types

We can assign a `Dog` and `Cat` object to the base type `Animal`, through inheritance. Here `Animal` is the static type, and `Dog`/`Cat` are dynamic types.

12.3 Polymorphic Assignment

Polymorphic assignment decides which method implementation is invoked given a virtual method in an inherited class.

```
abstract class A
{
    public virtual void Hello()
    {
        Console.WriteLine("Hello");
    }
}

class B : A
{
    public override void Hello()
    {
        Console.WriteLine("Bonjour");
    }
}
class C : A { }

class D : B { }

static void Main()
{
    A b = new B(); // static type: A, dynamic type: B
    A c = new C(); // static type: A, dynamic type: C
    B d = new D(); // static type: B, dynamic type: D

    b.Hello(); // prints: "Bonjour"
    c.Hello(); // prints: "Hello"
    d.Hello(); // prints: "Bonjour"
}
```

The program attempts to use the current class's implementation of `Hello`, and if no override exists, the base class implementation is used.

12.4 Interfaces

Interfaces allow us to enforce method implementations onto a derived class. Here derived classes *implement* interfaces. Interfaces only contain method signatures that act as contracts so that all implementing classes must provide an implementation. Another advantage of using interfaces is that a class can implement multiple interfaces. An interface is declared using the **`interface`** keyword, and it does not require the **`abstract`** or **`virtual`** keywords. Due to this, any derived classes also do not need to use the **`override`** keyword. As a convention, interfaces in C# start with a capital I.

```
interface IAnimal
{
    public void Speak();
}
class Dog : IAnimal
{
    public void Speak()
    {
        Console.WriteLine("Woof");
    }
}
...
static void Main()
{
    IAnimal dog = new Dog();
    dog.Speak(); // prints: "Woof"
}
```

12.5 Interface Design

Interfaces allow us to decouple code which enables modular design by removing dependencies. From the principle of Interface Segregation, it is better to have multiple simple interfaces rather than one large interface, so that clients can depend only on parts they require. By using interfaces, clients only see what each class does, rather than how it does it.

12.6 Future Expansion

When applying polymorphism, it is important to consider future expansion of types so that if a user wishes to implement a new derived type, fields and methods are carefully chosen to apply to many different problems. This includes the names given to class members.

12.7 Liskov Substitution Principle

The Liskov Substitution Principle is based on the concept of "substitutability". Given that `B` inherits from `A`, let `F(A a)` be a property that accepts objects of type `A`, then the property `F(B b)` should not alter any of the desirable properties of `F`. When inheriting from a class, we must ensure that all fields and methods are reusable and a genuine *is-a* relationship exists.