Programming Principals

Semester 1, 2022

Dr Alan Woodley

Tarang Janawalkar

Contents

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, input, and output
- 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 the 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. However, in a script, we must first compile the program to an executable in order to perform any tasks.

Definition 2.2 (Type)**.** The type of an expression is "what 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 integer 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 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.

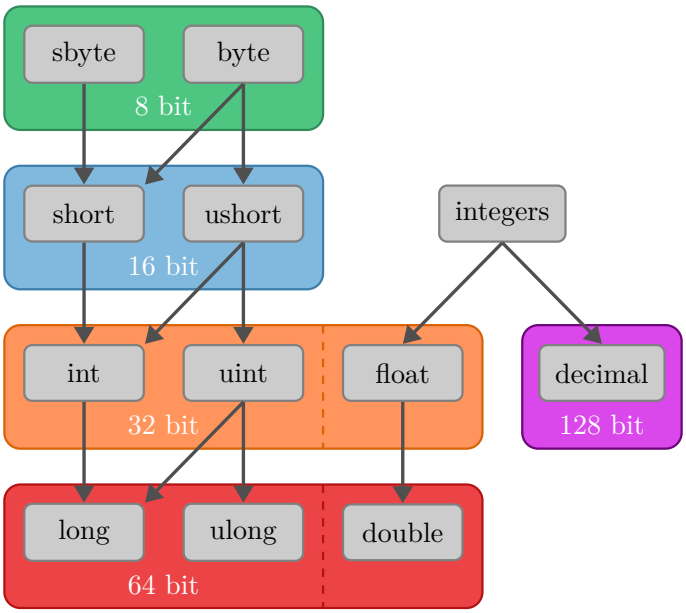

Figure 1: Numeric type implicit conversions in C#.

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 $1 < i < n$ 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. Goto Step 1.

3.3.2 Do-while Statements

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

Fundamental semantics:

1. Execute `STATEMENTS`.
2. Goto 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. Goto 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 Booleans

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 an integer, and is separated by a comma. 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, 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, the argument is generally passed by value.
The parameter type specifies what type of argument is to be passed. The parameter identifier gives the argument an alias within the method's scope, and 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 in macOS and Linux distributions, we must ensure that we are working in the correct directory.

6.2.1 Windows DOS

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