Chapter: 1

Introduction

1.1. The event driven life cycle in windows programming

Event-driven programming (EDP) is a programming paradigm where external events determine the flow of program execution. These events come in different shapes: user actions (e.g., button clicks, keyboard inputs), system events (like a finished file download), messages from other programs, sensor outputs, etc.

Event-driven programming is a programming pattern in which you create diagram code that executes outside the dataflow structure as a result of an event occurring.

Cont.’

Use event-driven programming when you want diagram code to execute in response to an event.

- For example, a single left-button mouse-click on a command button in a GUI program may trigger a routine that will open another window, save data to a database or exit the application.

1.2. The .Net framework and the Common Language Specification

What is .NET ?

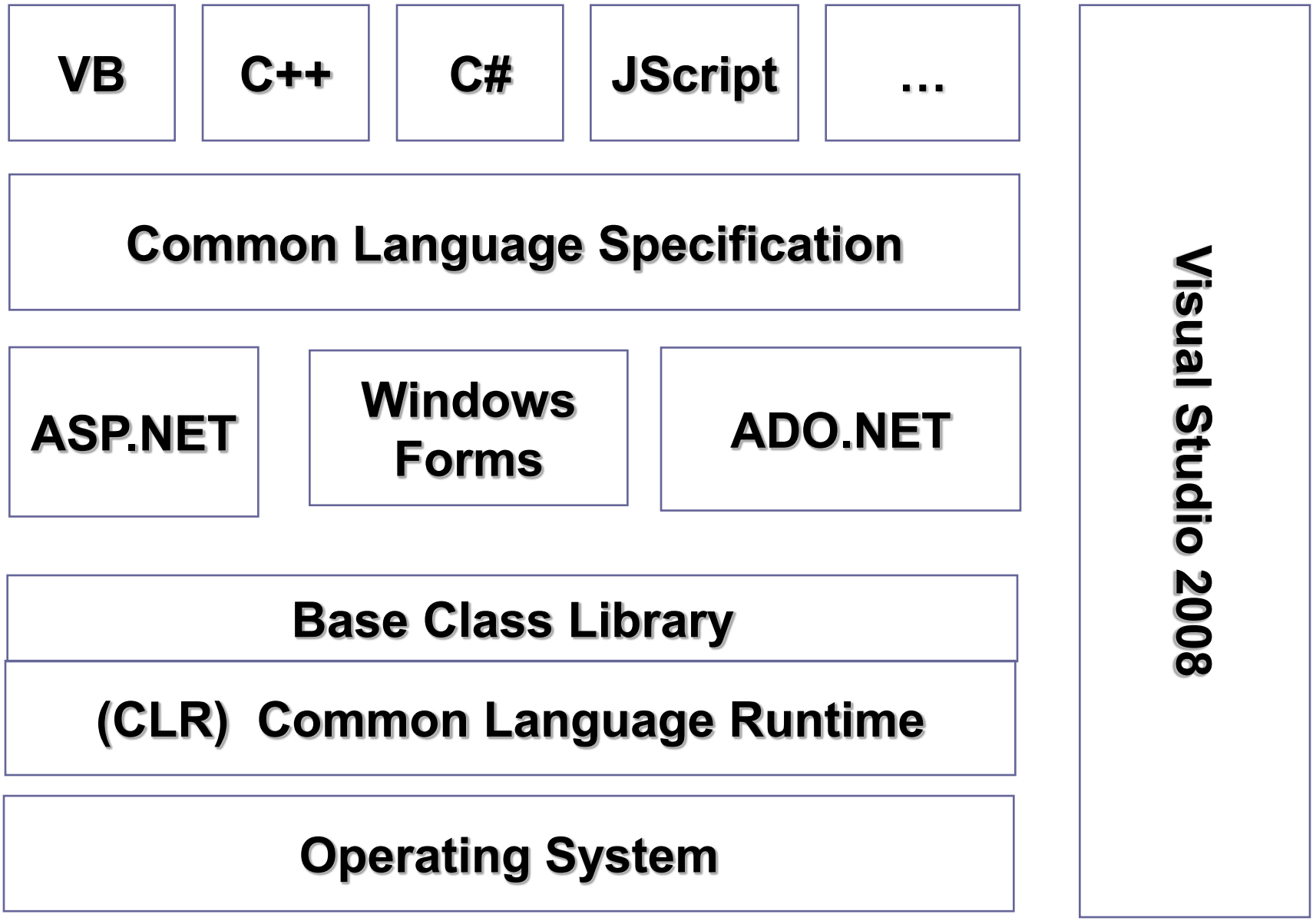
Microsoft.NET is a Framework

- Microsoft .NET is a Framework which provides a common platform to Execute or, Run the applications developed in various programming languages.
- Microsoft announced the .NET initiative.
- The main intention was to bridge the gap in **interoperability** between services of various programming languages.

.NET Framework Objectives

- **The .NET Framework is designed to fulfill the following objectives:**
  - Provide object-oriented programming environment
  - Provide environment for developing various types of applications, such as Windows-based applications and Web-based applications
  - To ensure that code based on the .NET Framework can integrate with any other code

.NET Framework



- **The .NET Framework consists of:**

– **The Common Language Specification (CLS)**

It contains guidelines, that language should follow so that they can communicate with other .NET languages. It is also responsible for Type matching.

– **The Framework Base Class Libraries (BCL)**

A consistent, object-oriented library of prepackaged functionality and Applications.

– **The Common Language Runtime (CLR)**

A language-neutral development & execution environment that provides common runtime for application execution .

Common Language Specification

CLS performs the following functions:

- Establishes a framework that helps enable cross-language integration, type safety, and high performance code execution
- Provides an object-oriented model that supports the complete implementation of many programming languages.
- Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.

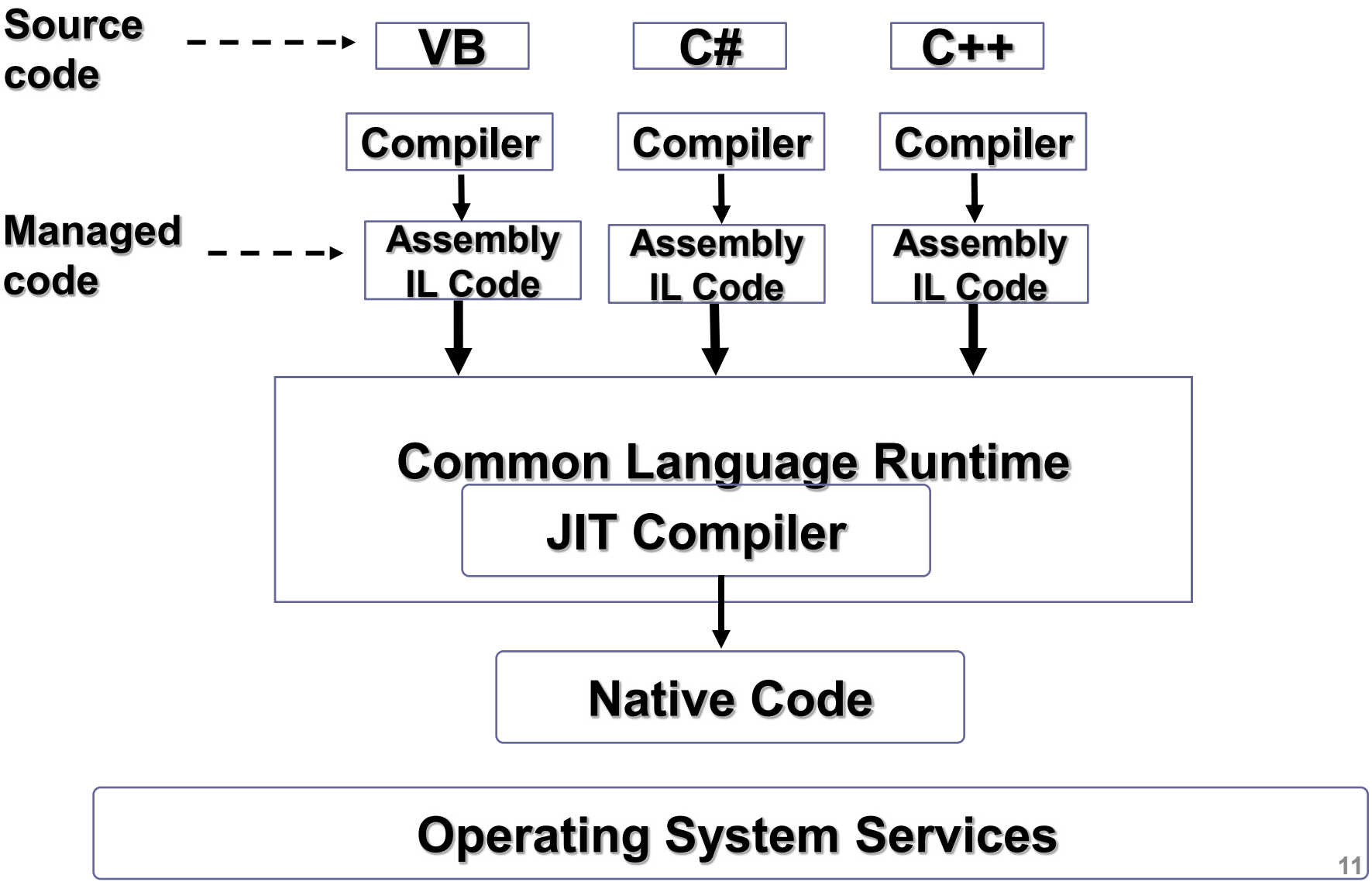
.NET Framework Base Class Library

- The Class Library is a complete, object-oriented collection of reusable types.
- These class library can be used to develop applications that include:
  - Traditional command-line applications
  - Graphical user interface (GUI) applications
  - Applications based on the latest innovations provided by ASP.NET
    - Web Forms
    - XML Web services

Common Language Runtime (CLR)

- CLR ensures:
  - A common *runtime* environment for all .NET languages
  - Uses *Common Type System (strict-type & code-verification)*
  - Memory allocation and garbage collection.
  - Security and interoperability of the code with other languages

Execution in CLR



Visual Studio IDE

Microsoft has introduced **Visual Studio.NET**, which is a tool (also called Integrated Development Environment)for developing .NET applications by using programming languages such as **VB, C# and VC++.** etc.

Cont.’

1.3. **Ms-Visual Studio IDE and basic features**

- Visual Studio is a powerful developer tool that you can use to complete the entire development cycle in one place. It is a comprehensive integrated development environment (**IDE**) that you can use to write, edit, debug, and build code, and then deploy your app
- To download vb software write”code.visualstudio.com/download” the in your browser

Ms-Visual Studio basic features

- VB is a GUI-based development tool that offers a faster RAD (Rapid Application Development or RAD means an adaptive software development model based on prototyping and quick feedback with less emphasis on specific planning) than most other programming languages.
- VB also features syntax that is more straightforward than other languages,
- a visual environment that is easy to understand and high database connectivity.

- Assignments on what is the event driven life cycle in windows programming? and what are in this cycle?

Cha

pter Two –Part I

Programming in C#

C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by Ecma and ISO.

C# was developed by Anders Hejlsberg and his team during the development of .Net Framework.

C# is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages to be used on different computer platforms and architectures.
The following reasons make C# a widely used professional language:
-Modern, general-purpose programming language

-Object oriented.

-Component oriented.

- Easy to learn.

-Structured language.

-It produces efficient programs.

-It can be compiled on a variety of computer platforms.

-Part of .Net Framework.

Strong Programming Features of C#

Although C# constructs closely follow traditional high-level languages, C and C++ and being an object-oriented programming language, it has strong resemblance with Java, it has numerous strong programming features that make it endearing to multitude of programmers worldwide.

C# Environment

C# is part of .Net framework and is used for writing .Net applications. The .Net framework is a revolutionary platform that helps you to write the following types of applications:

-Windows applications
-Web applications
-Web services

The .Net framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: C#, C++, Visual Basic, Jscript, COBOL, etc. All these languages can access the framework as well as communicate with each other.

The .Net framework consists of an enormous library of codes used by the client languages like C#. Following are some of the components of the .Net framework:

-Common Language Runtime (CLR)
-The .Net Framework Class Library
-Common Language Specification
-Common Type System
-Metadata and Assemblies
-Windows Forms

-ASP.Net and ASP.Net AJAX

- ADO.Net

- Windows Workflow Foundation (WF)

- Windows Presentation Foundation

- Windows Communication Foundation (WCF)

- LINQ

C# Program Structure

C# Hello World Example

A C# program basically consists of the following parts:

-Namespace declaration

- A class

- Class methods

- Class attributes

- A Main method

- Statements & Expressions

- Comments

Example

```
using System;
namespace HelloWorldApplication
{
 class HelloWorld
{
 static void Main(string[] args)
```

```
{
 /* my first program in C# */
 Console.WriteLine("Hello World");
 Console.ReadKey();
 }
}
}
```

-The first line of the program using System; - the using keyword is used to include the System namespace in the program. A program generally has multiple using statements.

-The next line has the namespace declaration. A namespace is a collection of classes. The HelloWorldApplication namespace contains the class HelloWorld.

- The next line has a class declaration, the class HelloWorld contains the data and method definitions that your program uses. Classes generally would contain more than one method. Methods define the behavior of the class. However, the HelloWorld class has only one method Main.

- The next line defines the Main method, which is the entry point for all C# programs. The Main method states what the class will do when executed

- The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program.

- The Main method specifies its behavior with the statement Console.WriteLine ("Hello World");WriteLine is a method of the Console class defined in the System namespace. This statement causes the message "Hello, World!" to be displayed on

the screen.

- The last line Console.ReadKey(); is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

It's worth to note the following points:

-C# is case sensitive.

- All statements and expression must end with a semicolon (;).

- The program execution starts at the Main method.

- Unlike Java, file name could be different from the class name.

C# Basic Syntax

C# is an object-oriented programming language. In Object-Oriented Programming methodology, a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or, more often, are said to be in the same class.

For example, let us consider a Rectangle object. It has attributes like length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating area and display details.

```
using System;

namespace RectangleApplication

{

 class Rectangle

 {
```

```
// member variables
double length;
double width;
public void Acceptdetails()
{
length = 4.5;
width = 3.5;
}
public double GetArea()
{
return length * width;
}
public void Display()
{
Console.WriteLine("Length: {0}", length);
Console.WriteLine("Width: {0}", width);
Console.WriteLine("Area: {0}", GetArea());
}
}
class ExecuteRectangle
```

```
{
 static void Main(string[] args)
{
 Rectangle r = new Rectangle();
r.Acceptdetails();
 r.Display();
 Console.ReadLine();
 }
 }
}
```

Comments in C#

Comments are used for explaining code. Compilers ignore the comment entries. The multiline comments in C# programs start with /* and terminates with the characters */ as shown below:

```
/* This program demonstrates
The basic syntax of C# programming
Language */
```

Single-line comments are indicated by the '//' symbol. For example,

```
}//end class Rectangle
```

Member Variables

Variables are attributes or data members of a class, used for storing data. In the preceding program, the Rectangle class has two member variables named length and width.

Member Functions

Functions are set of statements that perform a specific task. The member functions of a class are declared within the class. Our sample class Rectangle contains three member functions: AcceptDetails, GetArea and Display.

Instantiating a Class

In the preceding program, the class Execute Rectangle is used as a class, which contains the Main() method and instantiates the Rectangle class.

Identifiers

An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows:

-A name must begin with a letter that could be followed by a sequence of letters & underscore. The first character in an identifier cannot be a digit.

-It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : "' / and \. However, an underscore (_) can be used.

- It should not be a C# keyword.

Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers; however, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character. The following table lists the reserved keywords and contextual keywords in C#:

Reserved Keywords						
abstract	As	Base	bool	break	byte	case
catch	Char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	Goto	if	implicit	in	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	This	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	While					
Contextual Keywords						
add	Alias	ascending	descending	dynamic	from	get

C# Data Types

In C#, variables are categorized into the following types:

-Value type

- Pointer types

-Reference types

Type	Represents	Range	Defau Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / 10^{0 \text{ to } 28}$	0.0M
double	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324}$ to $(+/-)1.7 \times 10^{308}$	0.0D
float	32-bit single-precision floating point type	-3.4×10^{38} to $+ 3.4 \times 10^{38}$	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-923,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0

short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

Value Types

Value type variables can be assigned a value directly. They are derived from the The value types directly contain data. Some examples are int, char, float, which stores numbers, alphabets, floating point numbers, respectively. When you declare an int type, the system allocates memory to store the value.

To get the exact size of a type or a variable on a particular platform, you can use the size of method. The expression size of (type) yields the storage size of the object or type in bytes. Following is an example to get the size of int type on any machine:

```
namespace DataTypeApplication
{
 class Program
 {
 static void Main(string[] args)
 {
 Console.WriteLine("Size of int: {0}", sizeof(int));
```

```
 Console.ReadLine();
 }
}
}
```

Reference Types

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

In other words, they refer to a memory location. Using more than one variable, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value.

For example,

```
dynamic d = 20;
```

Dynamic types are similar to **object types** except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run-time.

STRING TYPE

The String Type allows you to assign any string values to a variable. The string type is an alias for the System. String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted. For example,

```
String str = "string type";
```

A @quoted string literal looks like:

@"stringtype";

C# Type Conversion

Type conversion is basically type casting or converting one type of data to another type. In C#, type casting has two forms:

1. Implicit type conversion - these conversions are performed by C# in a type-safe manner. Examples are conversions from smaller to larger integral types and conversions from derived classes to base classes.

2. Explicit type conversion - these conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator. The following example shows an explicit type conversion:

```
namespace TypeConversionApplication
{
 class ExplicitConversion
{
 static void Main(string[] args)
 {
 double d = 5673.74;
 int i;
 // cast double to int.
 i = (int)d;
 Console.WriteLine(i);
```

```
 Console.ReadKey();
 }
 }
}
```

C# Type Conversion Methods

C# provides the following built-in type conversion methods:

```
namespace TypeConversionApplication
{
 class StringConversion
 {
 static void Main(string[] args)
{
 int i = 75;
 float f = 53.005f;
 double d = 2345.7652;
bool b = true;
 Console.WriteLine(i.ToString());
 Console.WriteLine(f.ToString());
```

```
Console.WriteLine(d.ToString());

Console.WriteLine(b.ToString());

Console.ReadKey();

}

}

}
```

C# Variables

Avariable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Type.	Example
Integral typessbyte,	byte, short, ushort, int, uintlong, ulong&char
Floating point types.	float and double
Decimal types.	decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

Variable Definition in C#

Syntax for variable definition in C# is:

<data_type><variable_list>;

Here, data_type must be a valid C# data type including char, int, float, double, or any user-defined data type, etc., and variable_list may consist of one or more

identifier names separated by commas. Some valid variable definitions are shown here:

int i, j, k;

char c, ch;

float f, salary;

double d;

You can initialize a variable at the time of definition as:
int i = 100;

Variable Initialization in C#

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is:

variable_name = value;

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as:

<data_type><variable_name> = value;

Some examples are:

int d = 3, f = 5; /* initializing d and f. */

byte z = 22; /* initializes z. */

double pi = 3.14159; /* declares an approximation of pi. */

char x = 'x'; /* the variable x has the value 'x'. */

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.

Try the following example, which makes use of various types of variables:

```
namespace VariableDefinition
{
 class Program
{
 static void Main(string[] args)
 {
 short a;
 int b ;
 double c;
 /* actual initialization */
 a = 10;
 b = 20;
 c = a + b;
 Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
 Console.ReadLine();
}
 }
```

}

Accepting Values from User

The Console class in the System namespace provides a function ReadLine() for accepting input from the user and store it into a variable.

```
int num;
num = Convert.ToInt32(Console.ReadLIne());
```

The function Convert.ToInt32() converts the data entered by the user to int data type, because

Console.ReadLine() accepts the data in string format.

Lvalues and Rvalues in C#:

There are two kinds of expressions in C#:

1. lvalue: An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
2. rvalue: An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement

```
int g = 20;
```

But following is not a valid statement and would generate compile-time error:

```
10 = 20;
```

2.4. conditionals

Control Flow Statements

Now that we are familiar with condition statements, let us proceed to learn how we can use these statements to control the flow of a program.

If Statement

The if statement is one of the most commonly used control flow statements. It allows the program to evaluate if a certain condition is met, and to perform the appropriate action based on the result of the evaluation. The structure of an if statement is as follows (line numbers are added for reference):

if (condition 1 is met)

{

do Task A

}

else if (condition 2 is met)

{

do Task B

}

else if (condition 3 is met)

{

do Task C

}

else

```
{
 do Task E
}
```

```
example
if (20 > 18)
{
  Console.WriteLine("20 is greater than 18");
}
```

Switch Statement

The switch statement is similar to an if statement except that it does not work with a range of values. A switch statement requires each case to be based on a single value. Depending on the value of the variable used for switching, the program will execute the correct block of code.

The syntax of a switch statement is as follows:

switch (variable used for switching)

{

case firstCase:

do A;

break (or other jump statements);

case secondCase:

do B;

break (or other

do Task E

}

break (or other jump statements);

case default:

do C;

break (or other jump statements);

}

Example

C# Switch Statements

Use the `switch` statement to select one of many code blocks to be executed.

Syntax

```
switch(expression)
{
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
    break;
}
```

This is how it works:

- The `switch` expression is evaluated once
- The value of the expression is compared with the values of each `case`
- If there is a match, the associated block of code is executed
- The `break` and `default` keywords will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;
switch (day)
{
  case 1:
    Console.WriteLine("Monday");
    break;
  case 2:
    Console.WriteLine("Tuesday");
    break;
  case 3:
    Console.WriteLine("Wednesday");
    break;
  case 4:
    Console.WriteLine("Thursday");
    break;
  case 5:
    Console.WriteLine("Friday");
    break;
  case 6:
    Console.WriteLine("Saturday");
    break;
  case 7:
    Console.WriteLine("Sunday");
    break;
```

```
}

// Outputs "Thursday" (day 4)
```

The break Keyword

When C# reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

The default Keyword

The `default` keyword is optional and specifies some code to run if there is no case match:

Example

```
int day = 4;

switch (day)

{

  case 6:

    Console.WriteLine("Today is Saturday.");

    break;

  case 7:

    Console.WriteLine("Today is Sunday.");
```

```
    break;
  default:
    Console.WriteLine("Looking forward to the Weekend.");
    break;
}
// Outputs "Looking forward to the Weekend."
```

C# Break

You have already seen the `break` statement used in an earlier chapter of this tutorial. It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a **loop**.

This example jumps out of the loop when `i` is equal to `4`:

Example

```
for (int i = 0; i < 10; i++)
{
  if (i == 4)
  {
    break;
  }
  Console.WriteLine(i);
}
```

C# Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

Example

```
for (int i = 0; i < 10; i++)
{
  if (i == 4)
  {
    continue;
  }
  Console.WriteLine(i);
}
```

For Loop

The for loop executes a block of code repeatedly until the test condition is no longer valid.

The syntax for a for loop is as follows:

for (initial value; test condition; modification to value)

{

//Do Some Task

}

To understand how the for loop works, let's consider the example below.

```
for (int i = 0; i < 5; i++)
{
Console.WriteLine(i);
}
```

```
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

While Loop

Like the name suggests, a while loop repeatedly executes instructions inside the loop while a certain condition remains valid. The structure of a while statement is as follows:

while (condition is true)

{

do A

}

C# While Loop

The `while` loop loops through a block of code as long as a specified condition is `True`:

Syntax

```
while (condition)
{
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
int i = 0;
while (i < 5)
```

```
{
  Console.WriteLine(i);
  i++;
}
```

The Do/While Loop

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do
{
  // code block to be executed
}
while (condition);
```

The example below uses a `do/while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
int i = 0;
do
{
  Console.WriteLine(i);
  i++;
}
while (i < 5);
```

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

Break

The break keyword causes the program to exit a loop prematurely when a certain condition is met. We have already seen how the break keyword can be used in a switch statement. Now, let us look at an example of how the break keyword can be used in a for loop.

Consider the code segment below:

```
int i = 0;

for (i = 0; i < 5; i++)

{

Console.WriteLine("i = {0}", i);

if (i == 2)

break;

}
```

Continue

Another commonly used jump keyword is the continue keyword. When we use continue, the rest of the loop after the keyword is skipped for that iteration. An example will make it clearer.

If you run the code segment below

```
for (int i = 0; i<5; i++)

{

Console.WriteLine(“i = {0}”, i);

if (i == 2)

continue;

Console.WriteLine(“I will not be printed if i=2.\n”);

}
```

3.3. Arrays and collections

Arrays, Strings and Lists

Array

An array is simply a collection of data that are normally related to each other.

An array can be declared and initialized as follows:

int[] userAge = {21, 22, 23, 24, 25};

Array Properties and Methods

To use a property, we type the property name after the dot. To use a method, we type the method name after the dot operator, followed by a pair of parenthesis ().

Length

The Length property of an array tells us the number of items the array has.

For instance, if we have

int [] userAge = {21, 22, 26, 32, 40};

userAge.Length is equal to 5 as there are 5 numbers in the array.

Copy()

The Copy() method allows you to copy the contents of one array into another array, starting from the first element.

Suppose you have

int [] source = {12, 1, 5, -2, 16, 14};

and

int [] dest = {1, 2, 3, 4};

you can copy the first three elements of source into dest by using the statement below:

Array.Copy(source, dest, 3);

The first argument is the array that provides the values to be copied. The second is the array where the values will be copied into. The last argument specifies the number of items to copy.

In our example, our dest array becomes {12, 1, 5, 4} while the source array remains unchanged.

Sort()

The Sort() method allows us to sort our arrays. It takes in an array as the argument.

Suppose you have

int [] numbers = {12, 1, 5, -2, 16, 14};

You can sort this array by writing

Array.Sort(numbers);

The array will be sorted in ascending order. Thus, numbers becomes {-2, 1, 5, 12, 14, 16}.

String

A string is a piece of text. An example of a string is the text “Hello World”.

To declare and initialize a string variable, you write

string message = “Hello World”;

where message is the name of the string variable and “Hello World” is the string assigned to it. Note that you need to enclose the string in double quotes (“).

You can also assign an empty string to a variable, like this:

string anotherMessage = “”;

Finally, we can join two or more strings using the concatenate sign (+) and

String Properties and Methods

Length

The Length property of a string tells us the total number of characters the string contains.

To find the length of the string “Hello World”, we write

“Hello World”.Length

Substring()

The Substring() method is used to extract a substring from a longer string.

We can then use message to call the Substring() method

string newMessage = message.Substring(2, 5);

Substring(2, 5) extracts a substring of 5 characters from message, starting from index 2 (which is the third letter as indexes always start from 0).

Equals()

We can use the `Equals()` method to compare if two strings are identical.

If we have two strings as shown below

```
string firstString = "This is me";
string secondString = "Hello";

firstString.Equals("This is you");
```

returns `true` while

```
firstString.Equals(secondString);
```

returns false as the two strings (firstString and secondString) are not equal.

Lists

A list stores values like an array, but elements can be added or removed at will.

An array can only hold a fixed number of values. If you declare

int [] myArray = new int[10];

myArray can only hold 10 values. If you write myArray[10] (which refers to the 11th value since array index starts from zero), you will get an error.

If you need greater flexibility in your program, you can use a list.

To declare a list of integers, we write

List<int> userAgeList = new List<int>();

userAgeList is the name of the list.

List is a keyword to indicate that you are declaring a list.

The data type is enclosed in angle brackets < >.

You can choose to initialize the list at the point of declaration like this

List<int> userAgeList = new List<int> {11, 21, 31, 41};

To access the individual elements in a list, we use the same notation as when we access elements in an array. For instance, to access the first element, you write userAgeList[0]. To access the third element, you write userAgeList[2].

List Properties and Methods

The list data type also comes with a large number of properties and methods.

Add()

You can add members to a list using the Add() method.

userAgeList.Add(51);

userAgeList.Add(61);
userAgeList now has 6 members: {11, 21, 31, 41, 51, 61}.

Count

To find out the number of elements in the list, use the Count property.

userAgeList.Count gives us 6 as there are 6 elements in the list at the moment.

Insert()

To add members at a specific position, use the Insert () method.

To insert a member at the 3rd position, you write

userAgeList.Insert(2, 51);

where 2 is the index and 51 is the value you want to insert.

userAgeList now becomes {11, 21, 51, 31, 41, 51, 61}.

Remove()

To remove members from the list, use the Remove() method. The Remove() method takes in one argument and removes the first occurrence of that argument. For instance, if we write

userAgeList.Remove(51);

userAgeList becomes {11, 21, 31, 41, 51, 61}. Only the first ‘51’ is removed.

RemoveAt()

To remove a member at a specific location, use the RemoveAt() method.

For instance, to remove the 4th item (index 3), you write

userAgeList.RemoveAt(3);

where 3 is the index of the item to be removed.

userAgeList now becomes {11, 21, 31, 51, 61}.

Contains()

To check if a list contains a certain member, use the Contains() method.

To check if userAgeList contains ‘51’, we write

userAgeList.Contains(51);

We will get true as the result.

Clear()

To remove all items in a list, use the Clear() method. If we write

userAgeList.Clear();

we will have no elements left in the list.

Chapter 3: String and Date time data types

3.1. Manipulation with string data types

In C#, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h' , 'e' , 'l' , 'l' , and 'o' . We use the string keyword to create a string.Example

String in c# example Here is an example of using operators:

```
     int a = 7 + 9;
Console.WriteLine(a); // 16
string firstName = "John";
string lastName = "Doe";
// Do not forget the space between them
string fullName = firstName + " " + lastName;
Console.WriteLine(fullName); // John Doe
```

3.2. Working with date time types.

Date and time data types are used for values that contain date and time.

Date Data Type

The date data type is used to specify a date.

The date is specified in the following form "YYYY-MM-DD" where:

- YYYY indicates the year
- MM indicates the month
- DD indicates the day

Note: All components are required!

The following is an example of a date declaration in a schema

```
//parameters mean:(year ,month,day,hour,minute,seconds)
```

```
        var date01 = new DateTime(2008, 5, 1, 8, 30, 52);
            Console.WriteLine(date01)
```

An other formate

```
//Date time properties
        var date02 = DateTime.Now;
        var date03 = DateTime.UtcNow;//UTC = universal time coordinated
        var date04 = DateTime.Today;
        Console.WriteLine(date02);
        Console.WriteLine(date03);
        Console.WriteLine(date04);
        //Date time Methods
        //Adding or subtracting Date /time
        var tomorrow = date02.AddDays(1);
        Console.WriteLine(tomorrow);
        // Date Time format
        Console.WriteLine(date02.ToLongDateString());
        Console.WriteLine(date02.ToShortDateString());
        Console.WriteLine(date02.ToLongTimeString());
        Console.WriteLine(date02.ToShortTimeString());
        //custom date time format
        Console.WriteLine(date02.ToString());
        Console.WriteLine(date02.ToString("yy-MM-dd"));
        Console.ReadLine();
```

```
        }
    }
}
```

3.3. Arrays and collections

C# Arrays

Create an Array

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

```
string[] cars;
```

We have now declared a variable that holds an array of strings.

To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in **cars**:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Console.WriteLine(cars[0]);

// Outputs Volvo
```

Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

cars[0] = "Opel";

Console.WriteLine(cars[0]);

// Now outputs Opel instead of Volvo
```

Array Length

To find out how many elements an array has, use the `Length` property:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Console.WriteLine(cars.Length);
```

```
// Outputs 4
```

Loop Through an Array

You can loop through the array elements with the `for` loop, and use the `Length` property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.Length; i++)

{

  Console.WriteLine(cars[i]);

}
```

The foreach Loop

There is also a `foreach` loop, which is used exclusively to loop through elements in an **array**:

Syntax

```
foreach (type variableName in arrayName)

{

  // code block to be executed

}
```

The following example outputs all elements in the **cars** array, using a `foreach` loop:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

foreach (string i in cars)

{

  Console.WriteLine(i);

}
```

The example above can be read like this: **for each** `string` element (called **i** - as in **i**ndex) in **cars**, print out the value of **i**.

If you compare the `for` loop and `foreach` loop, you will see that the `foreach` method is easier to write, it does not require a counter (using the `Length` property), and it is more readabl

Sort Arrays

There are many array methods available, for example `Sort()`, which sorts an array alphabetically or in an ascending order:

Example

```
// Sort a string

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Array.Sort(cars);

foreach (string i in cars)

{

  Console.WriteLine(i);

}
```

```
// Sort an int
int[] myNumbers = {5, 1, 8, 9};
Array.Sort(myNumbers);
foreach (int i in myNumbers)
{
  Console.WriteLine(i);
}
```

System.Linq Namespace

Other useful array methods, such as `Min`, `Max`, and `Sum`, can be found in the `System.Linq` namespace:

Example

```
using System;
using System.Linq;

namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] myNumbers = {5, 1, 8, 9};
```

```
      Console.WriteLine(myNumbers.Max());  // returns the largest value
      Console.WriteLine(myNumbers.Min());  // returns the smallest value
      Console.WriteLine(myNumbers.Sum());  // returns the sum of elements
    }
  }
}
```

You will learn more about other namespaces in a later chapter.

Other Ways to Create an Array

If you are familiar with C#, you might have seen arrays created with the `new` keyword, and perhaps you have seen arrays with a specified size as well. In C#, there are different ways to create an array:

```
// Create an array of four elements, and add values later
string[] cars = new string[4];

// Create an array of four elements and add values right away
string[] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements without specifying the size
string[] cars = new string[] {"Volvo", "BMW", "Ford", "Mazda"};
```

```
// Create an array of four elements, omitting the new keyword, and without
specifying the size

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

It is up to you which option you choose. In our tutorial, we will often use the last option, as it is faster and easier to read.

However, you should note that if you declare an array and initialize it later, you have to use the `new` keyword:

```
// Declare an array

string[] cars;



// Add values, using new

cars = new string[] {"Volvo", "BMW", "Ford"};



// Add values without using new (this will cause an error)

cars = {"Volvo", "BMW", "Ford"};
```

C# Exercises

Exercise:

Create an array of type `string` called `cars`.

```
[    ] [    ] = {"Volvo", "
```

C# Methods

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method is defined with the name of the method, followed by parentheses **()**. C# provides some pre-defined methods, which you already are familiar with, such as `Main()`, but you can also create your own methods to perform certain actions:

Example

Create a method inside the Program class:

```
class Program
{
  static void MyMethod()
  {
    // code to be executed
  }
```

```
}
```

Example Explained

- `MyMethod()` is the name of the method
- `static` means that the method belongs to the Program class and not an object of the Program class. You will learn more about objects and how to access methods through objects later in this tutorial.
- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

Note: In C#, it is good practice to start with an uppercase letter when naming methods, as it makes the code easier to read.

Call a Method

To call (execute) a method, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, `MyMethod()` is used to print a text (the action), when it is called:

Example

Inside `Main()`, call the `myMethod()` method:

```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
  MyMethod();
```

```
}


// Outputs "I just got executed!"
```

A method can be called multiple times:

Example

```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
  MyMethod();
  MyMethod();
  MyMethod();
}

// I just got executed!
// I just got executed!
// I just got executed!
```

C# Exercises

Exercise:

Create a method named `MyMethod` and call it inside `Main()`.

```
static void ____()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
____
}
```

C# Method Parameters

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `string` called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example

```
static void MyMethod(string fname)
{
  Console.WriteLine(fname + " Referes");
}

static void Main(string[] args)
{
  MyMethod("Hussen");
  MyMethod("Yirgalem");
  MyMethod("Haymanot");
}

// Hussen Referes
// Yirgalem Referes
// Haymanot Referes
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: `fname` is a **parameter**, while `hussen`, `Yirgalem` and `Haymanot` are **arguments**.

Default Parameter Value

You can also use a default parameter value, by using the equals sign (=). If we call the method without an argument, it uses the default value ("Norway"):

Example

```
static void MyMethod(string country = "Norway")
{
  Console.WriteLine(country);
}

static void Main(string[] args)
{
  MyMethod("Sweden");
  MyMethod("India");
  MyMethod();
  MyMethod("USA");
}

// Sweden
// India
// Norway
// USA
```

A parameter with a default value, is often known as an "**optional parameter**". From the example above, `country` is an optional parameter and `"Norway"` is the default value.

Multiple Parameters

You can have as many parameters as you like:

Example

```
static void MyMethod(string fname, int age)
{
  Console.WriteLine(fname + " is " + age);
}

static void Main(string[] args)
{
  MyMethod("Liam", 5);
  MyMethod("Jenny", 8);
  MyMethod("Anja", 31);
}

// Liam is 5
// Jenny is 8
// Anja is 31
```

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Return Values

The `void` keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as `int` or `double`) instead of `void`, and use the `return` keyword inside the method:

Example

```
static int MyMethod(int x)
{
  return 5 + x;
}

static void Main(string[] args)
{
  Console.WriteLine(MyMethod(3));
}

// Outputs 8 (5 + 3)
```

This example returns the sum of a method's **two parameters**:

Example

```
static int MyMethod(int x, int y)
{
```

```
  return x + y;
}

static void Main(string[] args)
{
  Console.WriteLine(MyMethod(5, 3));
}

// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

Example

```
static int MyMethod(int x, int y)
{
  return x + y;
}

static void Main(string[] args)
{
  int z = MyMethod(5, 3);
  Console.WriteLine(z);
}
```

```
// Outputs 8 (5 + 3)
```

Named Arguments

It is also possible to send arguments with the *key: value* syntax.

That way, the order of the arguments does not matter:

Example

```
static void MyMethod(string child1, string child2, string child3)
{
  Console.WriteLine("The youngest child is: " + child3);
}

static void Main(string[] args)
{
  MyMethod(child3: "John", child1: "Liam", child2: "Liam");
}

// The youngest child is: John
```

C# Method Overloading

With **method overloading**, multiple methods can have the same name with different parameters:

Example

```
int MyMethod(int x)

float MyMethod(float x)

double MyMethod(double x, double y)
```

Consider the following example, which have two methods that add numbers of different type:

Example

```
static int PlusMethodInt(int x, int y)

{

  return x + y;

}



static double PlusMethodDouble(double x, double y)

{

  return x + y;

}



static void Main(string[] args)

{

  int myNum1 = PlusMethodInt(8, 5);
```

```
  double myNum2 = PlusMethodDouble(4.3, 6.26);

  Console.WriteLine("Int: " + myNum1);

  Console.WriteLine("Double: " + myNum2);

}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the `PlusMethod` method to work for both `int` and `double`:

Example

```
static int PlusMethod(int x, int y)

{

  return x + y;

}


static double PlusMethod(double x, double y)

{

  return x + y;

}


static void Main(string[] args)

{

  int myNum1 = PlusMethod(8, 5);

  double myNum2 = PlusMethod(4.3, 6.26);

  Console.WriteLine("Int: " + myNum1);

  Console.WriteLine("Double: " + myNum2);
```

```
}
```