Session Two:
Overview of C# Programming

DigiPen Institute of Technology
5001 150th Ave NE, Redmond, WA 98052
Phone: (425) 558-0299
www.digipen.edu
5.7 Function Returning Value Example

6 Flow Control

6.1 The “while” Statement
6.2 The “do-while” Statement
6.3 The “for” Loop
6.4 The “foreach, in” Statement
6.5 The “jump” Statement
  6.5.1 break
  6.5.2 continue
  6.5.3 goto
6.5.4 return
6.6 The “switch” Statement
6.7 if-else

7 Classes

7.1 Definition of a Class
7.2 Class Declaration
7.3 Members of a Class
7.4 Constants
7.5 Fields
7.6 Methods
7.7 Properties
7.8 Events
7.9 Indexers
7.10 Operators
7.11 Instance Constructors
7.12 Static Constructors
7.13 Destructors

8 Arrays

8.1 Introduction
8.2 Array Types
8.3 Array Creation
8.4 Array Element Access
8.5 Array Members
8.6 Array Initializers

9 Structures

9.1 Definition of a Struct
9.2 Struct Declaration
9.3 Members of a Struct
9.4 Class and Struct Differences

10 Miscellaneous

10.1 C# 2.0
  10.1.1 The “System.Diagnostics” Namespace
  10.1.2 The “System.Drawing” Namespace
  10.1.3 The “System.Windows.Forms” Namespace
  10.1.4 The “System.Collections.Generic” Namespace
10.2 Generics
10.3 The “Exception” Class
10.4 DirectX
10.4.1 Direct3D
  10.4.2 DirectInput
  10.4.3 DirectSound
  10.4.4 DirectX.AudioVideoPlayback
10.4.5 The “Vector2” Class
10.4.6 The “Matrix” Class
1 C# Programming Overview

1.1 Introduction

C# is a new, simple programming language based on the older programming language called C++. The similarities between C# and C++ are easily notified. This chapter is an overview of the C# programming language. In this chapter we might use some instructions or expressions without explaining their meaning. These materials will be covered in subsequent chapters.

1.2 The Smallest C# Program

```csharp
static void Main( )
{
}
```

OR

```csharp
static void Main( ) { }
```

The main function is the entry point of a C# program. All standard C# programs start by executing the content of the main function. However, this program does not do anything because the main function does not have instructions.

Facts:

- An open and close parenthesis is placed after the function name.
- Parentheses are used to hold the function arguments.
- The type of the value returned by the function is specified before the function name.
- Void is a C# type specifying a typeless type.
- The body of the function is written within the function block specified by the open and close curly braces.
- The C# programming language is a free format language.

1.3 Displaying a Message

The canonical “hello, world” program can be written as follows:

```csharp
using System;
class Hello
{
    static void Main() {
        Console.WriteLine("hello, world");
    }
}
```

The source code for a C# program is typically stored in one or more text files with a file extension of .cs, as in hello.cs. Using the command-line compiler, such a program can be compiled with the command line directive

csc hello.cs

which produces an application named hello.exe. The output produced by this application when it is run is:

```
hello, world
```

Close examination of this program is illuminating:

- The using System; directive references a namespace called System. This namespace contains the Console class referred to in the Main method. Namespaces provide a hierarchical means of organizing the elements of one or more programs. A “using” directive enables unqualified use of the types that are
members of the namespace. The “hello, world” program uses `Console.WriteLine` as shorthand for `System.Console.WriteLine`. (For the sake of brevity, most examples in this specification omit the `using System;` directive.)

- The `Main` method is a member of the class `Hello`. The entry point for an application—the method that is called to begin execution—is always a static method named `Main`.
- The “hello, world” output is produced using a class library. The language does not itself provide a class library. Instead, it uses a class library that is also used by other programming languages.

### 1.4 Comments

Two forms of comments are supported: single-line comments and delimited comments. Single-line comments start with the characters `//` and extend to the end of the source line. Delimited comments start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines. Comments do not nest. The character sequences `/*` and `*/` have no special meaning within a `//` comment, and the character sequences `//` and `/*` have no special meaning within a delimited comment.

The example below includes a delimited comment.

```csharp
/* Hello, world program
   This program writes "hello, world" to the console
*/
class Hello
{
    static void Main()
    {
        Console.WriteLine("hello, world");
    }
}
```

The following example shows several single-line comments.

```csharp
// Hello, world program
// This program writes "hello, world" to the console
// class Hello // any name will do for this class
{
    static void Main() // this method must be named "Main"
    {
        Console.WriteLine("hello, world");
    }
}
```

### 1.5 Multiple Instructions Program

A function is a block of code written to perform a specific task (“function”) and packaged in a unit so that it can be executed at desired points in a program. Dividing a program into separate functions makes the program more manageable and easier to understand.
class hello
{
    static void Main()
    {
        DisplayHello(); DisplayWorld();
        DisplayHelloWorld();
    }

    /* Function definition */
    static void DisplayHello()
    {
        Console.WriteLine("Hello");
    }

    static void DisplayWorld()
    {
        Console.WriteLine("World");
    }

    static void DisplayHelloWorld()
    {
        Console.WriteLine("Hello World");
    }
}

Facts:

- The program contains three user functions.
- Functions need to be defined before being used or executed.
- The main function contains three function calls or three statements.
- User functions are declared outside the main function.
- Functions can be called many times.
- The program starts by executing the first instruction in Main, which is the DisplayHello function.
- When the DisplayHello function is called, the execution flow changes to the first instruction within the function definition of DisplayHello.
- When the last instruction (which is also the first instruction) of the function DisplayHello is executed, the execution returns to the instruction right after the function call, which is DisplayWorld.

1.6 Functions with Argument Returning a Value

A function contains the parameters and the return type. The parameters provide data needed by the function to do its job. For each parameter specify its name and data type; if no parameters are needed, specify the keyword void inside the parentheses. The return type specifies the data type of the data that is returned by the function. Some functions do not return any data; in this case specify the keyword void as the return type.

Example:
void foo();
void foo(int i);
int foo();
int foo(int i);
class hello
{
    static void Main()
    {
        Console.WriteLine("{0}", Add(3, 5));
    }

    /* Function definition */
    static int Add(int x, int y)
    {
        return x + y;
    }
}

Facts:

- `int` is a C# type specifying whole or integral numbers.
- `int` means integer.
- The function prototype specifies that the function takes two integer arguments and returns an integer.
- When a function has more than one argument, a comma is used to separate the arguments.
- When a function with arguments is called, the arguments are received as parameters by the function where the function's instructions are specified.
- When arguments are passed to the parameters, the order of the arguments is respected. In our case, `x` would be equal to 3 and `y` would be equal to 5.
- The last statement of the function definition returns the result of the arithmetic expression `x + y`.
- `x` and `y` are called variables. A variable is a name assigned to a data storage location.
- The variables `x` and `y` are defined in the parameter list of the function: `(int x, int y)`.
- In C#, a variable must be defined before it can be used.
- A variable definition specifies its name and type.
- The compiler uses the type in order to know how much memory to allocate.

### 1.7 Variables

class add
{
    static void Main()
    {
        int i, j;
        i = 3;
        j = 5;
        Console.WriteLine("{0}", Add(i, j));
    }

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

Facts:

- Two integer variables `i` and `j` are declared and defined.
- By declaration, we mean that the rest of the function main knows about the presence and type of `i` and `j`.
In other words, the scope and type of \( i \) and \( j \) is within the body of function main. Then, an assignment operator is used in order to assign the value 3 and 5 to \( i \) and \( j \) respectively. The function \( \text{Add} \) is used by having two variables as arguments, while in the previous example the arguments were constants.

### 1.8 User Input

An address is a number that is assigned to each byte in a computer’s memory. It is used to track where data and instructions are stored in memory. Each byte is assigned an address whether or not it is being used to store data.

**Facts:**

- Instead of using the assignment operator to assign value to the variables, the input function \( \text{ReadLine} \) is used.
- This is an arithmetic expression.

```csharp
class test {
    static void Main() {
        int i, j;
        Console.Write("Enter i: ");
        i = Int32.Parse(Console.ReadLine());
        Console.Write("Enter j: ");
        j = Int32.Parse(Console.ReadLine());
        Console.WriteLine(Add(i, j));
    }
    static int Add(int x, int y) {
        return x+y;
    }
}
```

**Output**

5.3851648071345

**Facts:**

- Four variables of type integer are declared, defined, and assigned a value.
  
  ```csharp
  int x1=2, y1=1, x2=7, y2=3;
  ```
  
- In C#, you can assign an initial value to a variable during the declaration.
- The function \( \text{Distance} \) takes four integer arguments and returns a value of type \( \text{double} \).
- The C# double type allows the variable to have a value with fraction.
- A value with fraction is made from an integral part followed by a decimal point followed by the precision.
The function Distance requires a square root calculation. The square root function Sqrt is defined in the namespace called Math. The function Distance declares and defines six variables: x1, y1, x2, y2, deltaX, and deltaY. Remember that variables declared inside a function are only available while executing the function. deltaX and deltaY are used to hold the difference between the first and the second point. The statement deltaX=x2-x1; subtract x1 from x2 then assign the result to deltaX. In C# arithmetic, the multiplication * operator is evaluated from left to right. The multiplication operator has a higher order of precedence than the addition operator. This is why deltaX*deltaX is evaluated first to 25. Next deltaY*deltaY is evaluated to 4. Then 25 and 4 are added evaluating to 29. Then the square root of 29 is evaluated to 5.3851648071345.

1.9 Conditional Expression

```csharp
class test
{
    static void Main()
    {
        int i;
        string s;
        Console.WriteLine("Enter a number then press return:");
        i = Int32.Parse(Console.ReadLine());
        if(i>100)
            Console.WriteLine("{0} is larger than 100", i);
        else
            Console.WriteLine("{0} is less or equal than 100",i);
    }
}
```

Output

```
Enter a number then press return: 60
60 is less or equal than 100
```

Facts:

- i>100 is the Boolean expression that evaluates to true or false.
- If the expression is true, then the statement (or the block of statements enclosed between an opening and a closing curly braces) following the condition is executed.
- If the expression is false then the statement following the condition is skipped.
- The statement following the else is executed only when the conditional expression is false.
- In other words, only one of the WriteLine statements will be executed.
- The conditional statement starts with the keyword if followed by an opening parenthesis, followed by a Boolean expression, followed by a closing parenthesis, followed by an instruction:

```
if(Boolean expression)
    instruction;
```

Notice that there is no semicolon after the closing parenthesis of the Boolean expression because the conditional statement has not ended yet.
1.10 Loops

```csharp
class test {
    static void Main()
    {
        int i;
        for (i=0; i<10; i=i+1)
            Console.Write("{0} ", i);

        Console.Write("\n");
        i=0;
        while (i<10)
        {
            Console.Write("{0} ", i);
            i=i+1;
        }
    }
}
```

Output

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Facts:

- The for loop form is:
  
  ```
  for (expression1; expression2; expression3)
  statement
  ```

- The loop is initialized through expression1 i=0;
- Expression2 specifies the test made before each iteration i<10;
- If expression2 is true, the statement Console.Write("{0} ", i); is executed, then expression 3 i=i+1 is executed.
- The loop iterates until expression2 is false.
- If expression2 is false, the for loop will exit, and the control is transferred to the statement following the statement.
- Expression3 is evaluated after each iteration.
- Any or all of the three for expressions may be omitted, but the semicolon must remain.
- The while loop form is:
  
  ```
  while (expression)
  statement
  ```

If expression i<10 is true, the statement is executed until expression becomes false. In our case the statement is made from a block enclosed between curly braces:

```csharp
{
    Console.WriteLine("{0} ", i);
    i=i+1;
}
```

If expression is false, the execution resumes at the following statement. In our case, the following statement is the end of the program. The expression is evaluated before the statement is executed. When the expression is false from the first time, the statement will never be executed.
### 1.11 One-Dimensional Array

```csharp
class test
{
    static void Main()
    {
        int i;
        int[] A = new int[10];
        for(i=0;i<10;i=i+1)
            A[i]=i;
        for(i=0;i<10;i=i+1)
            Console.Write("{0} ",A[i]);
    }
}
```

**Output**

```
0 1 2 3 4 5 6 7 8 9
```

**Facts:**

- It is a collection of variables of the same type that are referred to by the same name.
- In our case `int[] A = new int[10];` reserved 10 integers.
- Array elements are accessed through an index; in our case the index is `i`.
- The first element is accessed by index 0 `A[0]`.
- The highest address corresponds to the last element and it is accessed by index (total number of elements – 1); in our case it is `A[9]`.
- The amount of storage required to hold an array depends on the type and the total number of elements; in our case it is `10 * 4=40`, since each integer is 4 bytes.
- The C# compiler does not perform index range checking.
- The array element is accessed by indexing the array name.
- It is done by writing the index enclosed between brackets placed after the array name. `arrayName[index] A[i]=i;`

### 1.12 Structure

```csharp
class test
{
    struct point
    {
        public int x;
        public int y;
    }
    static void Main()
    {
        point p1, p2;
        p1.x=2; p1.y=1;
        p2.x=7; p2.y=3;
        Console.WriteLine(Distance(p1,p2)); /* will print 5.3851648071345 */
    }
    static double Distance(point p1,point p2)
    {
        int deltaX, deltaY;
        deltaX=p2.x - p1.x;
        deltaY=p2.y - p1.y;
        return Math.Sqrt(deltaX*deltaX + deltaY*deltaY);
    }
}
```

**Facts:**

- A structure is an object consisting of a sequence of named members of various types.
- It is a collection of variables referenced under one name.
• The collection of variables is logically related.
• It provides a convenient way in order to keep related information together.

A structure is declared by typing the keyword `struct`, followed by the structure name, followed by the structure members enclosed between curly braces; for example:

```csharp
struct point
{
    public int x, y;
}
```

Once a structure is declared, variables having the structure type could be declared by typing the structure name followed by the variable name; for example:

```csharp
int deltaX;
```

The dot “.” operator is used to access the structure member. First write the structure variable name, followed by a dot, followed by a member; for example:

```csharp
deltaX=p2.x - p1.x;
```
2 C# TYPES

2.1 VALUE TYPES

In C#, a value type can be either a struct or an enumeration. C# contains a set of predefined struct types called the simple types. These simple types are identified through reserved words. All value types implicitly inherit form a class called object. Also, no type can derive from a value type. It is not possible for a value type to be null (null means "nothing" or "no value"). Assigning a variable of a value type creates a copy of the value. This is different from assigning a variable of a reference type, which copies the reference and not the object identified by the reference.

2.1.1 bool

The bool type represents boolean quantities. There can be two possible values of type bool: true and false. There is no standard conversion between bool and other types. Such conversions are accomplished by comparing an integral value to zero or comparing an object to null. A boolean value can be assigned to a bool variable, for example:

```csharp
bool MyBool = true;
```

You can also assign an expression that evaluates to a bool variable, for example:

```csharp
bool b = (i > 66);
```

Conversions
No conversion exists between the bool type and other types. For example, the following if statement:

```csharp
int i = 101;
if (i)
{
}
```

is not allowed in C#.
To test an int type, you have to explicitly compare it to a value, as follows:

```csharp
int i = 13;
if (i == 13)
{
    // do something
}
```

Example:

In this example, you enter a character from the keyboard and the program checks if the input character is a letter.

```csharp
public class test
{
    static void Main()
    {
        Console.Write("Enter a character: ");
        char ch = (char) Console.Read();

        if (Char.IsLetter(ch))
            Console.WriteLine("It is an alphabetic character.");
    }
}
```
else
    Console.WriteLine("It is not an alphabetic character.");
}
}

Output
Enter a character: A
It is an alphabetic character.

2.1.2 BYTE

The "byte" keyword denotes an integral type that stores values ranging from 0 to 255. Its size is 8-bit. A byte can be declared and initialized as follows:

    byte b = 117;

In the preceding declaration, 117 is implicitly converted from int to byte. If the integer literal exceeds the range of byte, a compilation error will occur, as with the following assignment statement:

    byte c = a + b; // Error: conversion from int to byte

To fix this problem, use an explicit cast:

    byte c = (byte)(a + b); // OK

2.1.3 CHAR

"char" is used to declare a Unicode character in the range 0 to 65535. Unicode characters are 16-bit characters used to represent most of the written languages throughout the world. The following statement declares a char variable and initializes it with the character D:

    char c = 'D';

2.1.4 DECIMAL

"decimal" denotes a 128-bit data type. The decimal type has a greater precision and a smaller range than the floating-point type, which makes it suitable for financial and monetary calculations. To make sure that a numeric real number is treated as a decimal, use the suffix m or M:

    decimal dec = 710.88m;

Without the suffix m, the number is treated as a double, and the expression generates a compilation error.

Example:
public class test
{
    static void Main ()
    {
        decimal dec = 12.4m;
        int i = 33;
        Console.WriteLine(dec * i); // Here the result is converted to decimal
    }
}

Formatting Decimal Output
You can format the results by using the String.Format method, or through the Console.WriteLine method, which calls String.Format(). The currency format is specified using the standard currency format string "C" or "c.

Example:
In this example, the output is formatted using the currency format string.
public class test
```csharp
static void Main()
{
    decimal dec1 = 0.987m;
    decimal dec2 = 545335566m;
    Console.WriteLine("{0:C}", dec1);
    Console.WriteLine("{0:C}", dec2);
}
```

Output:
$1.00
$545335566.00

### 2.1.5 Double

"double" denotes a simple type that stores 64-bit floating-point values. By default, a real numeric literal on the right-hand side of the assignment operator is treated as a double. However, if you want an integer number to be treated as a double, use the suffix `d` or `D`.

```csharp
double x = 44D;
```

Numeric integral types and floating-point types can be mixed in an expression. In this case, the integral types are converted to floating-point types.

Example:
```csharp
class test
{
    static void Main()
    {
        float f = 8.66f;
        int i = 123;
        double d = 22.1E+2;
        Console.Write("{0}", f + i + d);
    }
}
```

### 2.1.6 Enum

"enum" is used to declare an enumeration, which is a distinct type consisting of a set of constants called the enumerator list. Every enumeration type has an underlying type, which can be any integral type except `char`. The default type of the enumeration elements is `int`. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. For example:

```csharp
enum WeekDays {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

In this enumeration, Sun is 0, Mon is 1, and so forth. Enumerators can have initializers overriding the default values, as the following example shows:

```csharp
enum WeekDays {Sun=1, Mon, Tue, Wed, Thu, Fri, Sat};
```

In this enumeration, the sequence starts from 1. An enum `En` has a default value, which is the value produced by the expression `(En)0`. The enumeration type specifies how much storage is allocated. However, an explicit cast is needed to convert from enum type to an integral type.

Example:

In this example, an enumeration, Days, is declared. Two enumerators are explicitly converted to `int` and assigned to `int` variables.
public class test
{
    enum WeekDays {Sun=1, Mon, Tue, Wed, Thu, Fri, Sat};
    static void Main()
    {
        int i1 = (int) WeekDays.Tue;
        int i2 = (int) WeekDays.Thu;
        Console.WriteLine("Tuesday is day {0}", i1);
        Console.Write("Thursday is day {0}", i2);
    }
}

Output:
Tuesday is day 3
Thursday is day 5

### 2.1.7 Float

“float" denotes a type that can store 32-bit floating-point values. A real numeric literal on the right-hand side of the assignment operator is treated by default as a double. Therefore, to initialize a float, use the suffix f or F, for example:

```csharp
float f = 68.77F;
```

You will get a compilation error if you do not use the suffix because you are attempting to store a double value into a floating point variable.

You can mix numeric integral types and floating-point types in an expression. In this case, the integral types are converted to floating-point types.

Example:
```csharp
class test
{
    static void Main()
    {
        int i = 14;
        float f = 68.25f;
        Console.Write("{0}", i-f);
    }
}
```

### 2.1.8 Int

“int” denotes an integral type that stores 32-bit values. It ranges from -2,147,483,648 to 2,147,483,647. The type “int” is declared and initialized like this:

```csharp
int i = 441;
```

### 2.1.9 Long

“long” denotes an integral type that stores 64-bit values. It ranges from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The type “long” is declared and initialized like this:

```csharp
long myLong = 23940043;
```

When an integer literal has no suffix, its type is the first of the following types in which its value can fit: int, uint, long, ulong. The suffix L can be used with the long type like this:

```csharp
long myLong = 990085665543L;
```

When you use the suffix L or l, the literal integer’s type is either long or ulong according to its size. A predefined implicit conversion exists from long to float, double, or decimal. In other cases, a cast must be
used. For example, the following statement will produce a compilation error without an explicit cast:

\[
\text{int } i = 21L;
\]

There is an implicit conversion from \text{sbyte}, \text{byte}, \text{short}, \text{ushort}, \text{int}, \text{uint}, \text{or char} to \text{long}. Also, there is no implicit conversion from floating-point types to long. For example, the following statement generates an error:

\[
\text{long } l = 31.23;
\]

2.1.10 \text{SBYTE}

“sbyte” denotes an integral type that stores signed 8-bit integer values, ranging from -128 to 127. An “sbyte” can be declared and initialized like this:

\[
\text{sbyte } \text{mySbyte} = 100;
\]

100 is implicitly converted from \text{int} to \text{sbyte}. If the integer literal exceeds the range of \text{sbyte}, a compiler error will occur. A predefined implicit conversion exists from \text{sbyte} to \text{short, int, long, float, double, or decimal}. Also, there is no implicit conversion from floating-point types to \text{sbyte}.

2.1.11 \text{SHORT}

“short” denotes an integral data type that stores signed 16-bit integer values, ranging from -32,768 to 32,767. A “short” is declared and initialized like this:

\[
\text{short } s = 30201;
\]

30201 is implicitly converted from \text{int} to \text{short}. If the integer literal does not fit into a short storage location, a compiler error will occur. A predefined implicit conversion exists from \text{short} to \text{int, long, float, double, or decimal}. You cannot implicitly convert non-literal numeric types of larger storage size to short. Also there is no implicit conversion from floating-point types to \text{short}.

2.1.12 \text{STRUCT}

A “struct” is a value type. It can contain constructors, constants, fields, methods, properties, indexers, operators, and nested types. The struct type is suitable for representing objects such as Point, Rectangle, and Color. When creating a struct object using the new operator, it gets created, and the appropriate constructor is called. Structs can be instantiated without using the new operator. If you do not use the new operator, the fields will remain unassigned and the object cannot be used until all of the fields are initialized. You cannot declare a class using the keyword struct. Classes and structs are semantically different. A struct is a value type, while a class is a reference type.

Example:

\begin{verbatim}
public struct Point
{
    public int x, y;
}
\end{verbatim}

Example:
This example creates a Point object without using the new operator.

\begin{verbatim}
public struct Point
{
    public int x, y;
}
\end{verbatim}

\begin{verbatim}
class test
{
    static void Main()
    {

digiPen Game Development C# Webcast Series - Session Two, Version 1.0
Copyright © 2005 DigiPen (USA) Corporation.
2.1.13 uint

"uint" denotes an integral type that stores unsigned 32-bit integer values, ranging from 0 to 4,294,967,295. An "uint" can be declared and initialized like this:

```csharp
uint myUint = 4294967288;
```

The suffix `u` or `U` can be used, like this:

```csharp
uint myUint = 112U;
```

If you use the suffix `U` or `u`, the literal type is determined to be either `uint` or `ulong` according to its size. In this example, it is `uint`. A predefined implicit conversion exists from `uint` to `long`, `ulong`, `float`, `double`, or `decimal`. For example:

```csharp
float myFloat = 4294967289;
```

Also, there exists a predefined implicit conversion from `byte`, `ushort`, or `char` to `uint`. Otherwise you must use a cast. There is no implicit conversion from floating-point types to `uint`.

2.1.14 ulong

The "ulong" keyword denotes an integral type that stores unsigned 64-bit integer values, ranging from 0 to 18,446,744,073,709,551,615. A "ulong" is declared and initialized like this:

```csharp
ulong myUlong = 92854775806;
```

When using `L` or `l` as a suffix, the type of the literal integer will be either `long` or `ulong` according to its size. There is an implicit conversion from `ulong` to `float`, `double`, or `decimal`, but there is no implicit conversion from `ulong` to any integral type.

2.1.15 ushort

"ushort" denotes an integral data type that stores unsigned 16-bit integer values, ranging from 0 to 65,535. "ushort" can be declared and initialized like this:

```csharp
ushort myUShort = 65535;
```

65535 is implicitly converted from `int` to `ushort`. A compiler error will occur if the integer literal exceeds the range of `ushort`. A predefined implicit conversion exists from `ushort` to `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`. Also, there is a predefined implicit conversion from `byte` or `char` to `ushort`. Otherwise a cast must be used. There is no implicit conversion from floating-point types to `ushort`.

### 2.2 Reference Types

A reference type is one of the following: class, interface, array, or delegate. A reference type value is a reference to an *instance* of the type. "null" is compatible with all reference types and indicates the absence of an instance.

#### Class Types

A class defines a data structure containing data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors), and nested types.

#### Object Types

The `object` class type is the ultimate base class of all other types. Every type in C# directly or indirectly derives from the `object` class type.

#### String Types
The `string` type inherits directly from class `object`.

Interface Types
An interface defines a contract. A class implementing an interface must adhere to its contract.

Array Types
An array is a data structure containing a number of variables, which are accessed through indices. The variables contained in an array are called the elements of the array. They are all of the same type, and this type is called the element type of the array.

2.3 `void`

When used as the return type for a method, `void` specifies that the method does not return a value. `void` is not allowed in a method’s parameter list. A method with no parameters and returning no value is declared as follows:

```csharp
void MyMethod();
```
3 Variables

3.1 Introduction

In C#, a variable represents a storage location. A variable has a type that determines what values can be stored in this variable. Because C# is a type-safe language, the C# compiler guarantees that values stored in variables are always of the appropriate type. The value of a variable is changed through the assignment operator. The value of a variable is also changed through the use of the ++ and -- operators. A variable must be definitely assigned before its value can be obtained: variables are either initially assigned or initially unassigned. An initially assigned variable has a well-defined initial value. An initially unassigned variable has no initial value.

3.2 Categories of Variables in C#

In C#, there are seven categories of variables: static variables, instance variables, array elements, value parameters, reference parameters, output parameters, and local variables. The following sections describe each of these categories.

3.2.1 Static Variables

When declaring a variable with the “static” keyword, it is called a static variable. The initial value of a static variable is the default value of the variable’s type. A static variable is initially assigned.

3.2.2 Instance Variables

A variable declared without the “static” keyword is called an instance variable. An instance variable of a class exists when a new instance of that class is created, and ceases to exist when there are no references to that instance and the instance’s destructor (if any) has executed. The initial value of an instance variable of a class is the default value of the variable’s type. An instance variable of a class is initially assigned.

3.2.3 Array Elements

The elements of an array exist when an array instance is created, and cease to exist when there are no references to that array instance. The initial value of each of the elements of an array is the default value of the type of the array elements. An array element is initially assigned.

3.2.4 Value Parameters

A parameter declared without a ref or out modifier is a value parameter. A value parameter is initially assigned.

3.2.5 Reference Parameters

A parameter declared with a ref modifier is a reference parameter. This represents the same storage location as the variable given as the argument in the function member invocation. Therefore, the value of a reference parameter is always the same as the underlying variable. A variable has to be definitely assigned before it can be passed as a reference parameter in a function member invocation. A reference parameter is considered initially assigned within a function member.
3.2.6 Output Parameters

An output parameter is a parameter declared with an `out` modifier. An output parameter represents the same storage location as the variable given as the argument in the function member invocation. Therefore, the value of an output parameter is always the same as the underlying variable. A variable does not need to be definitely assigned before it can be passed as an output parameter in a function member invocation. Within a function member, an output parameter is unassigned initially.

3.2.7 Local Variables

A local variable is declared within a block, a `for`-statement, a `switch`-statement, or a `using`-statement. The lifetime of a local variable is implementation-dependent. For example, the compiler could generate code that results in the variable’s storage having a shorter lifetime than its containing block. A local variable is not automatically initialized and has no default value. It also is unassigned initially. A compile-time error results if the local variable is referred a position that precedes its declaration.

3.3 Default Values

Static variables, instance variables of class instances, and array elements are automatically initialized to their default values. The default value of a variable depends on the type of the variable.
4 Expressions, Statements & Operators

What is an expression?

- It is a sequence of operators and operands.
- It specifies computation of a value, or
- It designates a variable or constant.

4.1 Classification of Expressions

An expression is classified as one of the following:

- A value.
- A variable.
- A namespace.
- A type.
- A method group.
- A property access.
- An event access.
- An indexer access.
- Nothing.

The final result of an expression cannot be one of the following:

- A namespace.
- A type.
- A method group.
- An event access.

These categories are intermediate constructs. They are only permitted in certain contexts.

Values of Expressions

If the expression denotes a property access, an indexer access, or a variable, the value of the property, indexer, or variable is implicitly substituted:

- The value of a variable is simply the value currently stored in the storage location identified by the variable.
- The value of a property access expression is obtained by invoking the get-accessor of the property.
- The value of an indexer access expression is obtained by invoking the get-accessor of the indexer.

4.2 Operators

Expressions are constructed from operands and operators. Operators of an expression indicate which operations to apply to the operands.

We can find three types of operators:

1. Unary Operators: The unary operators take one operand. They use either prefix notation (i--) or postfix notation (i++).
2. Binary Operators: The binary operators take two operands. They all use infix notation (i + j).
3. Ternary Operator: There exists only one ternary operator, ?: The ternary operator takes three operands and uses infix notation (x ? y : z).

Operands in an expression are evaluated from left to right. Certain operators can be overloaded. This permits user-defined operator implementations to be specified for operations where one or both of the operands are of a
user-defined class or struct type.

**Operator Precedence and Associativity**

When an expression contains multiple operators, the **precedence** of the operators controls the order in which the individual operators are evaluated. For example, the expression $a + b * c$ is evaluated as $a + (b * c)$ because the $*$ operator has higher precedence than the $+$ operator. The precedence of an operator is established by the definition of its associated grammar production.

When an operand occurs between two operators with the same precedence, the **associativity** of the operators controls the order in which the operations are performed:

- All binary operators are **left-associative**, except for the assignment operators, meaning that operations are performed from left to right.
- Assignment operators and conditional operator (?:) are **right-associative**, which means that operations are performed from right to left.
- Precedence and associativity can be controlled using parentheses.

**Operator Overloading**

User-defined implementations can be introduced by including **operator declarations** in classes and structs. User-defined operator implementations always take precedence over predefined operator implementations – only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered.

The **overloadable unary operators** are:

- $+ \quad - \quad ! \quad ~ \quad ++ \quad -- \quad true \quad false$

The **overloadable binary operators** are:

- $+ \quad - \quad * \quad / \quad \% \quad & \quad | \quad ^ \quad << \quad >> \quad == \quad != \quad > \quad < \quad >= \quad <=$

When a binary operator is overloaded, the corresponding assignment operator (if any) is also implicitly overloaded. For example, an overload of operator $+$ is also an overload of operator $+=$. The assignment operator ($=$) cannot be overloaded. An assignment performs a bit-wise copy of a value into a variable. Element access, such as $Ar[x]$, is not an overloadable operator.

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator. For example, the $/$ operator is always a binary operator, always has the precedence level specified in, and is always left-associative.

### 4.3 Function Members

Function members are members that contain executable statements. They are always members of types and cannot be members of namespaces.

**C#** defines the following categories of function members:

- Methods
- Properties
- Events
- Indexers
- User-defined operators
- Instance constructors
- Static constructors
- Destructors

Statements contained in function members are executed through function member invocations. The argument list of a function member invocation provides actual values or variable references for the parameters of the function member.

**The new Operator**

The `new` operator is used to create new instances of types. There are three forms of `new` expressions:
1. Object creation expressions are used to create new instances of class types and value types.
2. Array creation expressions are used to create new instances of array types.
3. Delegate creation expressions are used to create new instances of delegate types.

The `new` operator implies creation of an instance of a type. Instances of value types require no additional memory beyond the variables in which they reside.

### 4.4 Unary Operators

The unary operators are `+`, `-`, `!`, `~`, `*`, `++`, `--`, and cast operators.

#### The + Operator

The predefined plus operators are:

- `int operator +(int x);`
- `uint operator +(uint x);`
- `long operator +(long x);`
- `ulong operator +(ulong x);`
- `float operator +(float x);`
- `double operator +(double x);`
- `decimal operator +(decimal x);`

For each of these operators, the result is simply the value of the operand.

#### The - Operator

The predefined negation operators are:

1. Integer negation:
   - `int operator -(int x);`
   - `long operator -(long x);`
   The result is computed by subtracting `x` from zero.

2. Floating-point negation:
   - `float operator -(float x);`
   - `double operator -(double x);`
   The result is the value of `x` with its sign inverted.

3. Decimal negation:
   - `decimal operator -(decimal x);`
   The result is computed by subtracting `x` from zero. Decimal negation is equivalent to using the unary minus operator of type `Decimal`.

4. Logical negation operator:
   There is only one predefined logical negation operator:
   - `bool operator !(bool x);`
   This operator computes the logical negation of the operand: if the operand is `true`, the result is `false`. If the operand is `false`, the result is `true`.

5. Bitwise complement operator:
   The bitwise complement operators are:
   - `int operator ~(int x);`
   - `uint operator ~(uint x);`
   - `long operator ~(long x);`
   - `ulong operator ~(ulong x);`

6. Prefix increment and decrement operators:
• Pre-increment-expression: $$++ \text{ unary-expression}$$
• Pre-decrement-expression: $$-- \text{ unary-expression}$$

The value returned by the operator becomes the result of the operation. The $$++$$ and $$--$$ operators also support postfix notation.

Cast Expressions

A cast-expression is used to explicitly convert an expression to a given type.

Example:

$$(\text{type}) \text{ unary-expression}$$

4.5 Arithmetic Operators

The $$*, /, \%, +, -$$ operators are called the arithmetic operators.

multiplicative-expression:

unary-expression

multiplicative-expression $$*$$ unary-expression

multiplicative-expression $$/$$ unary-expression

multiplicative-expression $$\%$$ unary-expression

additive-expression:

multiplicative-expression

additive-expression $$+$$ multiplicative-expression

additive-expression $$-$$ multiplicative-expression

Multiplication Operator

Integer multiplication:

• int operator $$(\text{int } x, \text{ int } y);$$
• uint operator $$(\text{uint } x, \text{ uint } y);$$
• long operator $$(\text{long } x, \text{ long } y);$$
• ulong operator $$(\text{ulong } x, \text{ ulong } y);$$

Floating-point multiplication:

• float operator $$(\text{float } x, \text{ float } y);$$
• double operator $$(\text{double } x, \text{ double } y);$$

Decimal multiplication:

• decimal operator $$(\text{decimal } x, \text{ decimal } y);$$

Division Operator

Integer division:

• int operator $$(\text{int } x, \text{ int } y);$$
• uint operator $$(\text{uint } x, \text{ uint } y);$$
• long operator $$(\text{long } x, \text{ long } y);$$
• ulong operator $$(\text{ulong } x, \text{ ulong } y);$$

Floating-point division:

• float operator $$(\text{float } x, \text{ float } y);$$
• double operator $$(\text{double } x, \text{ double } y);$$

Decimal division:

• decimal operator $$(\text{decimal } x, \text{ decimal } y);$$

Remainder Operator
Integer remainder:
- int operator % (int x, int y);
- uint operator % (uint x, uint y);
- long operator % (long x, long y);
- ulong operator % (ulong x, ulong y);

Floating-point remainder:
- float operator %(float x, float y);
- double operator %(double x, double y);

Decimal remainder:
- decimal operator %(decimal x, decimal y);

Addition Operator

Integer addition:
- int operator +(int x, int y);
- uint operator +(uint x, uint y);
- long operator +(long x, long y);
- ulong operator +(ulong x, ulong y);

Floating-point addition:
- float operator +(float x, float y);
- double operator +(double x, double y);

Decimal addition:
- decimal operator +(decimal x, decimal y);

Enumeration addition:
- E operator +(E x, U y);
- E operator +(U x, E y);

String concatenation:
- string operator +(string x, string y);
- string operator +(string x, object y);
- string operator +(object x, string y);

Subtraction Operator

Integer subtraction:
- int operator -(int x, int y);
- uint operator -(uint x, uint y);
- long operator -(long x, long y);
- ulong operator -(ulong x, ulong y);

Floating-point subtraction:
- float operator -(float x, float y);
- double operator -(double x, double y);

Decimal subtraction:
- decimal operator -(decimal x, decimal y);

Enumeration subtraction:
- U operator -(E x, E y);
- E operator -(E x, U y);
4.6 Shift Operators

The << and >> operators are used to perform bit shifting operations.

\[
\text{shift-expression}:
\]

\[
\text{additive-expression}
\]

\[
\text{shift-expression} \ll \text{additive-expression}
\]

\[
\text{shift-expression} \gg \text{additive-expression}
\]

When declaring an overloaded shift operator, the type of the first operand must always be the class or struct containing the operator declaration, and the type of the second operand must always be \text{int}.

The predefined shift operators are listed below.

\text{Shift Left}
\begin{itemize}
\item \text{int operator} \ll (<\text{int x, int count});
\item \text{uint operator} \ll (<\text{uint x, int count});
\item \text{long operator} \ll (<\text{long x, int count});
\item \text{ulong operator} \ll (<\text{ulong x, int count});
\end{itemize}

The << operator shifts \text{x} left by a number of bits computed as described below. The high-order bits outside the range of the result type of \text{x} are discarded, the remaining bits are shifted left, and the low-order empty bit positions are set to zero.

\text{Shift Right}
\begin{itemize}
\item \text{int operator} \gg (>\text{int x, int count});
\item \text{uint operator} \gg (>\text{uint x, int count});
\item \text{long operator} \gg (>\text{long x, int count});
\item \text{ulong operator} \gg (>\text{ulong x, int count});
\end{itemize}

The >> operator shifts \text{x} right by a number of bits computed as described below.

4.7 Relational & Type Testing Operators

The ==, !=, <, >, <=, >=, is and as operators are called the relational and type testing operators.

\text{Relational-Expression}
\begin{itemize}
\item \text{shift-expression}
\item \text{relational-expression} < \text{shift-expression}
\item \text{relational-expression} > \text{shift-expression}
\item \text{relational-expression} <= \text{shift-expression}
\item \text{relational-expression} >= \text{shift-expression}
\item \text{relational-expression} is \text{type}
\item \text{relational-expression} as \text{type}
\end{itemize}

\text{Equality-Expression}
\begin{itemize}
\item \text{relational-expression}
\item \text{equality-expression} == \text{relational-expression}
\item \text{equality-expression} != \text{relational-expression}
\end{itemize}

All comparison operators return a result of type \text{bool}, as described in the following table:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{x == y}</td>
<td>true if \text{x} is equal to \text{y}, false otherwise</td>
</tr>
<tr>
<td>\text{x != y}</td>
<td>true if \text{x} is not equal to \text{y}, false otherwise</td>
</tr>
<tr>
<td>\text{x &lt; y}</td>
<td>true if \text{x} is less than \text{y}, false otherwise</td>
</tr>
</tbody>
</table>
The as Operator
This operator is used to explicitly convert a value to a given reference type using a reference conversion or a boxing conversion. The as operator never throws an exception. Instead, if the indicated conversion is not possible, the resulting value is null.

### 4.8 Logical Operators

The &, ^, and | operators are called the logical operators.

- **and-expression:**
  - equality-expression
  - and-expression & equality-expression
- **exclusive-or-expression:**
  - and-expression
  - exclusive-or-expression ^ and-expression
- **inclusive-or-expression:**
  - exclusive-or-expression
  - exclusive-or-expression |
  - exclusive-or-expression

### Conditional Operator

The ?: operator is called the conditional operator. It is sometimes called the ternary operator. A conditional expression of the form \( b \? x : y \) first evaluates the condition \( b \). Then, if \( b \) is true, \( x \) is evaluated and becomes the result of the operation. Otherwise, \( y \) is evaluated and becomes the result of the operation. A conditional expression never evaluates both \( x \) and \( y \). The conditional operator is right-associative, meaning that operations are grouped from right to left. The first operand of the ?: operator must be an expression of a type that can be implicitly converted to bool, or an expression of a type that implements operator true. If neither requirement is satisfied, a compile-time error occurs. The second and third operands of the ?: operator control the type of the conditional expression.

### Assignment Operators

Assignment operators assign a new value to a variable, a property, or an indexer element. The left operand of an assignment must be an expression classified as a variable, a property access, or an indexer access. The = operator is called the simple assignment operator. It assigns the value of the right operand to the variable, property, or indexer element given by the left operand. The operators formed by prefixing a binary operator with an = character are called the compound assignment operators. These operators perform the indicated operation on the two operands, and then assign the resulting value to the variable, property, or indexer element given by the left operand. The assignment operators are right-associative, meaning that operations are grouped from right to left. For example, an expression of the form \( a = b = c \) is evaluated as \( a = (b = c) \).

### Simple Assignment

The – operator is called the simple assignment operator. In a simple assignment, the right operand must be an expression of a type that is implicitly convertible to the type of the left operand. The operation assigns the value of the right operand to the variable, property, or indexer element given by the left operand. The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value. If the left operand is a property or indexer access, the property or indexer must have a set accessor. If this is not the case, a compile-time error occurs.

### Boolean Expressions

A boolean-expression is an expression that yields a result of type bool. The controlling conditional expression of an if-statement, while-statement, do-statement, or for-statement is a boolean-expression. The controlling conditional expression of the ?: operator follows the same rules as a boolean-expression, but for reasons of
operator precedence is classified as a *conditional-or-expression*. A *boolean-expression* is required to be of a type that can be implicitly converted to `bool` or of a type that implements `operator true`. If neither requirement is satisfied, a compile-time error occurs. When a boolean expression is of a type that cannot be implicitly converted to `bool` but does implement `operator true`, then following evaluation of the expression, the `operator true` implementation provided by that type is invoked to produce a `bool` value.
5 Functions

5.1 Definition

The definition of a function includes the return type, the function name, the parameters’ list, and the function body. The function body is enclosed between an opening and a closing brace. Example:

```csharp
int add(int a, int b)
{
    return a + b;
}
```

The parameters $a$ and $b$ receive the values of the arguments when the function is called. Example:

```csharp
int n;
 n = add(4,5);
```

5.2 Scope

The code found in a function is private to the function and cannot be accessed by any statement from another function. Function code is accessed only through function call. A C# program starts from the function `Main`. All functions have a file scope. Parameters and variables declared inside the function have function scope. They are created when the function is entered and destroyed when the function ends. Static variables declared inside the function have a function scope, but they retain their values between function calls.

5.3 Calling a Function

A function is called by using its name followed by an opening and closing parenthesis including the arguments, if any. If the function has no arguments, then the argument list is left empty. When a function is called, the function parameters receive the value of the arguments by respecting their order. Parameter1 receives the value of argument1, and ParameterN receives the value of argumentN.

When a function is called, the sequential execution flow is interrupted, and the program counter jumps to the first statement of the function called. Once the last statement is executed, or a return instruction is executed, the sequential execution flow is interrupted and the program counter is updated so it points to the next instruction after the function call. The return instruction or statement also returns a value since the function has a returning type. In other words, the function could be used as value specified by its type.

5.4 “return” Statement Example

```csharp
class test
{
    struct box
    {
        public float left;
        public float top;
        public float right;
        public float bottom;
    }

    static float Maximum(float v1, float v2)
    {
```
if(v1>v2)
    return v1;
else
    return v2;
}

static float Minimum(float v1, float v2)
{
    if(v1<v2)
        return v1;
    else
        return v2;
}

static bool Intersect(float v1, float v2)
{
    if(v1-v2<=0)
        return true;
    else
        return false;
}

static void Main()
{
    box b1, b2;
    Console.WriteLine("Enter the left, top, right, and bottom coordinates of b1:");
    /* the input from keyboard is stored in the structure members */
    b1.left = Int32.Parse(Console.ReadLine());
    b1.top = Int32.Parse(Console.ReadLine());
    b1.right = Int32.Parse(Console.ReadLine());
    b1.bottom = Int32.Parse(Console.ReadLine());
    Console.WriteLine("Enter the left, top, right, and bottom coordinates of b2:");
    b2.left = Int32.Parse(Console.ReadLine());
    b2.top = Int32.Parse(Console.ReadLine());
    b2.right = Int32.Parse(Console.ReadLine());
    b2.bottom = Int32.Parse(Console.ReadLine());
    if(Intersect(Maximum(b1.left, b2.left)-Minimum(b1.right, b2.right),
                Minimum(b1.bottom, b2.bottom)-Maximum(b1.top, b2.top)))
        Console.WriteLine("b1 and b2 intersect\n");
    else
        Console.WriteLine("b1 and b2 do not intersect\n");
}

### 5.5 Call by Value Function Arguments

Arguments can be passed in two ways: Call by value and call by reference. The call by value method copies the value of the argument into the function parameter. Therefore, changes made to the parameter have no effect on the argument. In other words, the function code cannot modify the arguments used to call the function.

Example:

class test
{
    static void Main()
```csharp
{   int x;
    x = 10;
    callByValue(x);
  }

static void callByValue(int x)
{
    x = 20;
}

5.6 Call by Reference Function Arguments

To pass a parameter by reference, use the ref or out keyword. A variable of a reference type does not contain its data directly; it contains a reference to its data. When you pass a reference-type parameter by value, it is possible to change the data pointed to by the reference, such as the value of a class member. However, you cannot change the value of the reference itself; that is, you cannot use the same reference to allocate memory for a new class and have it persist outside the block. To do that, pass the parameter using the ref (or out) keyword. Example:

```csharp
class test
{
    static void Main()
    {
        int x = 10, y = 20;
        Console.WriteLine("Before calling swap: x={0}, y={1}", x, y);
        /* call by reference arguments*/
        swap(ref x, ref y);
        Console.WriteLine("After calling swap: x={0}, y={1}", x, y);
    }
}

/* call by reference parameters*/
static void swap(ref int p1, ref int p2)
{
    int tmp;
    tmp = p1; /*Saving p1 in tmp*/
    p1 = p2; /*p1 takes the value of p2*/
    p2 = tmp; /*p2 takes the value of tmp*/
}
5.7 Function Returning Value Example

class test
{
    static void Main()
    {
        int i;
        Console.Write("Enter an integer value: ");
        i = Int32.Parse(Console.ReadLine());
        Console.WriteLine("The absolute value of \{0\} is \{1\}", i, Absolute(i));
    }

    /* the function receives an integer and returns its absolute value */
    static int Absolute(int i)
    {
        return i>=0? i:-i;
    }
}
6 Flow Control

6.1 The “while” Statement

The while statement executes a block of statements repeatedly until a specified expression evaluates to false. It has the form while (expression) statement where expression is an expression that can be implicitly converted to bool or a type that contains overloading of the true and false operators. The expression is used to test the loop-termination criteria. statement is the statement(s) that will be executed.

A while loop executes zero or more times because the test of expression takes place before each execution of the loop. A while loop can terminate when a break, goto, return, or throw statement transfers control outside the loop. To pass control to the next iteration without exiting the loop, use the continue statement.

Example:
using System;
class Test
{
    static void Main()
    {
        int i = 16;
        while (n > 0)
        {
            Console.WriteLine("Another value of i: {0}", i);
            i -= 4;
        }
    }
}

6.2 The “do-while” Statement

The do statement executes a block of statements repeatedly until a specified expression evaluates to false. It is in the following form:

do statement while (expression);

where expression is an expression that can be implicitly converted to bool. statement is the statement(s) to be executed. Regardless of the value of expression, the body of the do statement is executed at least once.

Example:
using System;
public class Test
{
    static void Main()
    {
        int i = 16;
        do
        {
            Console.WriteLine("Another value of i: {0}", i);
            i -= 4;
        }
        while(i > 0);
    }
}

Example:
using System;
public class Test
{
    static void Main ()
    {
        int i = 16;
        do
        {
            Console.WriteLine("Another value of i: {0}", i);
            i -= 4;
        }
        while(i > 18);
    }
}

In the preceding example, although the condition evaluates initially to false, the loop will be executed once.

6.3 The "for" Loop

The for loop executes a block of statements repeatedly until a specified expression evaluates to false. It is in the following form:

   for ([initializers]; [expression]; [iterators]) statement

where initializers is a comma separated list of expressions or assignment statements that will initialize the loop, expression is an expression that can be implicitly converted to bool, iterators is an expression statement(s) that increment or decrement the loop counters, and statement is the embedded statement(s) to be executed.

The for statement executes the statement repeatedly as follows: First, the initializers are evaluated. Then, while the expression evaluates to true, the statement(s) are executed. When expression evaluates to false, control is transferred outside the loop.

A for statement executes zero or more times because the test of expression takes place before the execution of the loop. All of the expressions of the for statement are optional.

Example:
using System;
public class Test
{
    static void Main()
    {
        for (int i = 6; i >0; i--)
        {        
            Console.Write(i);
        }
    }
}

6.4 The "foreach, in" Statement

The foreach statement repeats a group of statements for each element in an array. The foreach statement is used to iterate through the array to get the desired information, but should not be used to change the contents of the array. The statement is in the following form:

   foreach (type identifier in expression) statement

where type is the type of identifier, identifier is the iteration variable that represents the array element, expression is the array expression, and statement is the statement(s) to be executed.
The statements continue to execute for each element in the array. Control is transferred to the next statement following the `foreach` block after the iteration has been completed for all the elements. When used with an array, the `foreach` statement repeats the embedded statement(s) for each element in the array.

Example:
```csharp
using System;
class test {
    static void Main() {
        int[] ar = new int[] {0,-1,2,-3,4,-5,6,-7,8,-9};
        foreach (int i in ar) {
            if (i < 0)
                Console.WriteLine("This is a negative number");
            else
                Console.WriteLine("This is a positive number");
        }
    }
}
```

6.5 The “jump” Statement

The jump statement is used for branching, which causes a transfer of the program control. The following keywords are used in jump statements: `break, continue, goto, return`.

6.5.1 break

The `break` statement terminates the closest enclosing loop or conditional statement in which it appears. Control is passed to the statement that follows the terminated statement, if any. The `break` statement takes the form: `break;

Example:
```csharp
using System;
class Test {
    static void Main() {
        for (int i = 15; i >= 0; i--)
            if (i == 10)
                break;
            Console.Write(i);
    }
}
```

6.5.2 continue

The `continue` statement passes control to the next iteration of the enclosing iteration statement in which it appears. It takes the form: `continue;`

Example:
```csharp
using System;
class Test {
    static void Main() {
        for (int i = 15; i >= 0; i--)
            if (i == 10)
                break;
            Console.Write(i);
    }
}
```
for (int i = 13; i > 1; i--)
{
    if (i > 2)
        continue;
    Console.WriteLine("The current value is : {0}", i);
}

6.5.3 goto

The goto statement transfers control directly to a labeled statement. It can be one of the following forms:

goto identifier;
goto case constant-expression;
goto default;

where identifier is a label and constant-expression is switch-case label.

- identifier indicates a label located in the current body, the same scope, or an enclosing scope of the goto statement.
- goto is commonly used to transfer control to a switch-case label.
- goto is useful to get out of nested loops.
- A warning message can be issued if the label has never been referenced in the program.

Example:
using System;
class test
{
    static void Main()
    {
        int i = 0;
        do
            {
                if (i == 8) goto Label1;
                else i++;
            }
        while (true);

        Label1: Console.Write(i);
    }
}

6.5.4 return

The return statement terminates execution of the method in which it appears and returns control to the calling method. If the method is of the type void, the return statement can be omitted. The return statement has the form:
return [expression];

where expression is the value returned by a method. expression is not used with methods of the type void.

Example:
class Test
{
    static int AddInt(int i1, int i2)
{ 
    int j = i1 + i2;
    return j;
}

public static void Main()
{
    int n1 = 120;
    int n2 = 125;
    int sum = AddInt(n1,n2);
}

6.6 The “switch” Statement

The switch statement selects for execution a statement from a list. The type of a switch statement is specified by the switch expression. The switch statement can have at most one default label.

A switch statement is executed as follows:

- The switch expression is evaluated and converted to the specified type.
- If one of the constants specified in a case label is equal to the value of the switch expression, control is transferred to the statement list following the matched case label.
- If none of the constants specified in case labels is equal to the value of the switch expression, control is transferred to the statement list following the default label, if a default label exists.
- If no default label is present, control is transferred outside the switch statement.

A switch section is not permitted to “fall through” to the next switch section. When execution of a switch section is to be followed by execution of another switch section, a goto case or goto default statement must be used. Multiple labels are permitted in a switch section.

The statement list of a switch section typically ends in a break, goto case, or goto default statement. The governing type of a switch statement may be the type string. If this is the case, null is permitted as a case label constant. The statement list is reachable if the switch statement is reachable and at least one of the following is true:

1. The switch expression is a non-constant value.
2. The switch expression is a constant value that matches a case label in the switch section.
3. The switch expression is a constant value that does not match any case label, and the switch section contains the default label.
4. A switch label of the switch section is referenced by a reachable goto case or goto default statement.

The end point is reachable if at least one of the following is true:
1. The switch statement contains a reachable break statement that exits the switch statement.
2. The switch statement is reachable, the switch expression is a non-constant value, and no default label is present.
3. The switch statement is reachable, the switch expression is a constant value that does not match any case label, and no default label is present.

6.7 if-else

The if statement is a control statement that executes a block of code if an expression evaluates to true. It has the form:

if (expression)
    statement1
where expression is an expression that can be converted to bool, statement1 is the statement(s) to be executed if expression is true, and statement2 is the statement(s) to be executed if expression is false. If expression is true, statement1 is executed. If the optional else clause exists and expression evaluates to false, statement2 is executed. After executing the if statement, control is transferred to the next statement.

If any of the two results of the if statement (true or false) results in executing more than one statement, multiple statements can be conditionally executed by including them into blocks. The statement(s) to be executed upon testing the condition can be of any kind, including another if statement nested into the original if statement. In nested if statements, the else clause belongs to the last if that does not have a corresponding else. ill be displayed if the condition \((x > 10)\) evaluates to false.

Example:
```csharp
using System;
public class Test
{
    static void Main()
    {
        Console.Write("Enter something from the keyboard ");
        char ch = (char) Console.Read();
        if (Char.IsLetter(ch))
            if (Char.IsLower(ch))
                Console.WriteLine("What you entered is a lowercase character.");
            else
                Console.WriteLine("What you entered is an uppercase character.");
        else
            Console.WriteLine("What you entered is not a character.");
    }
}
```

Example:
```csharp
using System;
public class Test
{
    static void Main()
    {
        Console.Write("Press a key: ");
        char ch = (char) Console.Read();

        if (Char.IsUpper(ch))
            Console.WriteLine("You pressed an uppercase character.");
        else if (Char.IsLower(c))
            Console.WriteLine("You pressed a lowercase character.");
        else if (Char.IsDigit(c))
            Console.WriteLine("You pressed a digit.");
        else
            Console.WriteLine("What You pressed is not alphanumeric.");
    }
}
```
7 Classes

7.1 Definition of a Class

A class is a data structure. It may contain data, functions, and nested types. Data members include constants and fields. Function members include methods, operators, events, properties, indexers, instance constructors, destructors and static constructors. A class support inheritance, which is a mechanism that allows a derived class to extend and specialize a base class.

7.2 Class Declaration

A class-declaration is a type-declaration that declares a new class. It consists of an optional set of attributes, followed by an optional set of class-modifiers, followed by the keyword class and an identifier that names the class, followed by an optional class-base specification, followed by a class-body, and optionally followed by a semicolon.

Class Modifiers

A class-declaration may optionally include a sequence of class modifiers: new, public, protected, internal, private, abstract, and sealed. In a class declaration, if the same modifier appears multiple times, it results in a compiler-time error. The new modifier is permitted on nested classes. It specifies that the class hides an inherited member by the same name. If the new modifier appears on a class declaration that is not a nested class declaration, it results in a compile-time error. The public, protected, internal, and private modifiers control the accessibility of the class.

Abstract Classes

The abstract modifier indicates that a class is incomplete and that it is intended only as a base class. An abstract class cannot be instantiated directly. Using the new operator on an abstract class is a compile-time error. An abstract class can contain abstract members and it cannot be sealed.

Sealed Classes

The sealed modifier prevents derivation from a class. If a sealed class is specified as the base class of another class, it generates a compile-time error. A sealed class cannot be an abstract class.

Base Classes

When a class-type is included in the class-base, it specifies the direct base class of the class being declared. If a class declaration has no class-base, the direct base class is assumed to be “object.” A class inherits members from its direct base class.

Example:

class B {}
class D: B {}

Class B is said to be the direct base class of D, and D is said to be derived from B. Class B implicitly derives from “object.”

The direct base class of a class type must be at least as accessible as the class type itself. For example, it is a compile-time error for a public class to derive from a private or internal class. The base classes of a class are the direct base class and its base classes. Except for class object, every class has exactly one direct base class. The object class has no direct base class and is the ultimate base class of all other classes.

7.3 Members of a Class

Class members consist of the members introduced by its class-member-declarations and the members inherited
from the direct base class. Class members are divided into the following categories:

1. Fields are the class variables.
2. Constants represent constant values associated with the class.
3. Methods implement the computations and actions that can be performed by the class.
4. Properties define characteristics associated with reading and writing those characteristics.
5. Indexers permit instances of the class to be indexed like arrays.
6. Events define notifications that can be generated by the class.
7. Instance constructors implement class initialization.
8. Operators define the expression operators that would be applied to instances of the class.
9. Static constructors implement the actions required to initialize the class itself.
10. Destructors implement the actions to be performed before instances of the class are deleted.
11. Types represent local types of the class.

Members that can contain executable code are known as the function members of the class. The function members of a class are the events, operators, methods, properties, indexers, instance constructors, static constructors, and destructors of that class.

A class-declaration creates a new declaration space, and the class-member-declarations immediately contained by the class-declaration introduce new members into this declaration space. Rules that apply to class-member-declarations are:

1. Instance constructors, destructors and static constructors should have the same name as the immediately enclosing class.
2. The name of a constant, property, type, field, or event should differ from the names of all other members declared in the same class.
3. The name of a method should differ from the names of all other non-methods declared in the same class.
4. The signature of a method should differ from the signatures of all other methods declared in the same class.
5. The signature of an instance constructor should differ from the signatures of all other instance constructors declared in the same class.
6. The signature of an indexer should differ from the signatures of all other indexers declared in the same class.
7. The signature of an operator should differ from the signatures of all other operators declared in the same class.
8. The inherited members of a class are not part of the declaration space of a class. Therefore, a derived class is allowed to declare a member with the same name or signature as an inherited member.

Inheritance
A class inherits the members of its direct base class. It implicitly contains all members of its direct base class, except for the instance constructors, destructors and static constructors of the base class. Inheritance is transitive. If C is derived from B, and B is derived from A, then C inherits the members declared in B as well as the members declared in A.
A derived class extends its direct base class. It can add new members to those it inherits, but it cannot remove the definition of an inherited member. Instance constructors, destructors, and static constructors are not inherited. A derived class can hide inherited members by declaring new members with the same name or signature.
A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation vary depending on the run-time type of the instance through which the function member is invoked.

Access Modifiers
It is a compile-time error to specify more than one access modifier, except for the protected internal combination. When a class-member-declaration does not include any access modifiers, private is assumed. A class-member-declaration can have any one of the five possible kinds of declared accessibility: public, protected internal, protected, internal, or private.

Static and Instance Members
Members of a class are either **static members** or **instance members**. Static members belong to classes, and instance members belong to objects (instances of classes). When a method, event, field, property, operator, or constructor declaration includes a static modifier, it declares a static member. Additionally, a constant or type declaration implicitly declares a static member. When a method, event, field, property, indexer, constructor, or destructor declaration does not include a static modifier, it declares an instance member.

**Nested Types**
A type declared within a class or struct is called a **nested type**. A type that is declared within a compilation unit or namespace is called a **non-nested type**.

**Remark:** The `this` keyword within a nested type cannot be used to refer to instance members of the containing type.

**Access to Private and Protected Members of the Containing Type**
A nested type has access to all of the members that are accessible to its containing type, including members of the containing type that have private and protected declared accessibility.

**Reserved Member Names**
For each member declaration that is a property, event, or indexer, the implementation must reserve two method signatures based on the kind of the member declaration, its name, and its type. It is a compile-time error for a program to declare a member whose signature matches one of these reserved signatures. The reserved names do not introduce declarations, thus they do not participate in member lookup. Destructor declaration causes a signature to be reserved.

For a property P of type T, the following signatures are reserved:

- T get_P();
- void set_P(T value);

Both signatures are reserved, even if the property is read-only or write-only.

For an event E of delegate type T, the following signatures are reserved:

- void add_E(T handler);
- void remove_E(T handler);

For an indexer of type T with parameter-list L, the following signatures are reserved:

- T get_Item(L);
- void set_Item(L, T value);

Both signatures are reserved, even if the indexer is read-only or write-only.

For a class containing a destructor, the following signature is reserved:

- void Finalize();

### 7.4 Constants

A **constant** is a class member that represents a constant value that can be computed at compile-time. A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type. Constants are permitted to depend on other constants within the same program as long as the dependencies are not of a circular nature.

### 7.5 Fields

A **field** represents a variable associated with an object or class. A **field-declaration** introduces one or more fields of a given type. It declares that multiple fields are the same as multiple declarations of single fields with the same attributes, modifiers, and type.
Static and Instance Fields
When a field declaration includes a static modifier, the fields introduced are static fields. When no static modifier is present, the fields introduced are instance fields. A static field is not part of a specific instance. There is only one copy of a static field for the associated application domain. An instance field belongs to an instance. Every instance of a class contains a separate set of all instance fields of the class.

Readonly Fields
When a field-declaration includes a readonly modifier, the fields are readonly fields. Direct assignments to readonly fields can only occur as part of the declaration or in an instance constructor (for readonly non-static fields) or static constructor (for readonly static fields) in the same class. Attempting to assign to a readonly field or passing it as an out or ref parameter in any other context results in a compile-time error.

7.6 Methods

A method is a member that implements a computation or action that can be performed by an object or class. Methods are declared using method-declarations. The return-type of a method declaration specifies the type of the value computed and returned by the method. The return-type is void if the method does not return a value. The member-name specifies the name of the method.

Method Parameters
The method’s formal-parameter-list declares the parameters of a method, if any exist.

Value Parameters
A parameter declared with no modifiers is a value parameter. It corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation. When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression of a type that is implicitly convertible to the formal parameter type. A method is permitted to assign new values to a value parameter.

Reference Parameters
A parameter declared with a ref modifier is a reference parameter. It does not create a new storage location. A reference parameter represents the same storage location as the variable given as the argument in the method invocation. A variable must be definitely assigned before it can be passed as a reference parameter. Within a method, a reference parameter is always considered definitely assigned.

Output Parameters
A parameter declared with an out modifier is an output parameter. It does not create a new storage location. Output parameters represent the same storage location as the variable given as the argument in the method invocation. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned. Within a method an output parameter is considered initially unassigned and must be definitely assigned before its value is used. Every output parameter of a method must be definitely assigned before the method returns.

Static and Instance Methods
When a method declaration includes a static modifier, the method is said to be a static method. When no static modifier is present, the method is said to be an instance method. A static method does not operate on a specific instance, and it is a compile-time error to refer to this in a static method. On the other hand, an instance method operates on a given instance of a class, and this instance can be accessed as this.

7.7 Properties

A property is a member that provides access to a characteristic of an object or a class. The length of a string, the caption of a window, the name of a customer, and the size of a font are all examples of a property. Properties are a natural extension of fields and do not denote storage locations. The type of a property declaration specifies the type of the property introduced by the declaration, and the member-name specifies the name of the property. The type of a property must be at least as accessible as the property itself.
**Static and Instance Properties**
When a property declaration includes a static modifier, the property is said to be a **static property**. When no static modifier is present, the property is said to be an **instance property**. A static property is not associated with a specific instance. On the other hand, an instance property is associated with a given instance of a class, and this instance can be accessed as `this` in the accessors of the property.

**Virtual, Sealed, Override, and Abstract Accessors**
A virtual property declaration specifies that the accessors of the property are virtual. The virtual modifier applies to both accessors of a read-write property. It is not possible for only one accessor of a read-write property to be virtual.

An abstract property declaration specifies that the accessors of the property are virtual, but it does not provide an actual implementation of the accessors. Non-abstract derived classes are required to provide their own implementation for the accessors by overriding the property. A property declaration that includes both the abstract and override modifiers specifies that the property is abstract and overrides a base property. Abstract property declarations are only permitted in abstract classes. The accessors of an inherited virtual property can be overridden in a derived class by including a property declaration that specifies an override directive. An overriding property declaration may include the sealed modifier. The accessors of a sealed property are also sealed. Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods.

**7.8 Events**
An **event** is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying **event handlers**. An event can be used as the left hand operand of the `+=` and `-=` operators.

**Static and Instance Events**
When an event declaration includes a static modifier, the event is said to be a **static event**. When no static modifier is present, the event is said to be an **instance event**. A static event is not associated with a specific instance. On the other hand, an instance event is associated with a given instance of a class, and this instance can be accessed as `this` in the accessors of the event.

**7.9 Indexers**
An **indexer** is a member that enables an object to be indexed in the same way as an array. An indexer element is not classified as a variable; therefore, it is not possible to pass an indexer element as a `ref` or `out` argument. The formal parameter list of an indexer defines the signature of the indexer, which consists of the number and types of its formal parameters. The element type and names of the formal parameters are not part of an indexer’s signature. The signature of an indexer must differ from the signatures of all other indexers declared in the same class. Indexers and properties are conceptually similar, but they differ in many ways. When an indexer declaration includes an extern modifier, the indexer is said to be an **external indexer**.

**7.10 Operators**
An **operator** is a member that defines the meaning of an expression operator that can be applied to instances of the class. There are three categories of overloadable operators:

1. Unary operators.
2. Binary operators.
3. Conversion operators.

An operator declaration must include both a public and a static modifier. When an operator declaration includes an extern modifier, the operator is said to be an **external operator**. For all non-external operators, the operator-body consists of a `block` which specifies the statements to execute when the operator is invoked.

The parameter(s) of an operator must be value parameters. The signature of an operator must differ from the signatures of all other operators declared in the same class. All types referenced in an operator declaration
must be at least as accessible as the operator itself. When the same modifier appears multiple times in an operator declaration, it results in a compile-time error. Each operator category imposes additional restrictions, as described in the following sections. Like other members, operators declared in a base class are inherited by derived classes.

### 7.11 Instance Constructors

An **instance constructor** is a member that implements the actions required to initialize an instance of a class. **Constructor-declarations** declare instance constructors. A constructor-declaration may include a set of attributes, a valid combination of the four access modifiers, and an extern modifier. A constructor declaration is not permitted to include the same modifier multiple times.

The identifier of a constructor-declarator must name the class in which the constructor is declared. Specifying any other name results in a compile-time error. The formal parameter list defines the signature of an instance constructor and governs the process whereby overload resolution selects a particular instance constructor in an invocation. Each of the types referenced in the formal-parameter-list of an instance constructor must be at least as accessible as the constructor itself. The optional constructor-initializer specifies another instance constructor to invoke before executing the statements given in the constructor-body of this instance constructor.

When a constructor declaration includes an extern modifier, the constructor is said to be an **external constructor**. Because an external constructor declaration provides no actual implementation, its constructor-body consists of a semicolon. For all other constructors, the constructor-body consists of a block, which specifies the statements to initialize a new instance of the class. This corresponds exactly to the block of an instance method with a void return type.

Instance constructors are not inherited. Thus, a class has no instance constructors other than those actually declared in the class. If a class contains no instance constructor declarations, a default instance constructor is automatically provided. Instance constructors are invoked by object-creation-expressions and through constructor-initializers.

**Default Constructors**

If a class contains no instance constructor declarations, a default instance constructor is automatically provided. The default constructor simply invokes the parameterless constructor of the direct base class. If the direct base class does not have an accessible parameterless instance constructor, a compile-time error occurs. If the class is abstract, then the declared accessibility for the default constructor is protected. Otherwise, the declared accessibility for the default constructor is public. A default constructor is provided because the class contains no instance constructor declarations.

### 7.12 Static Constructors

A **static constructor** is a member that implements the actions required to initialize a class. Static constructors are not inherited, and cannot be called directly. The exact timing of static constructor execution is implementation-dependent.

### 7.13 Destructors

A **destructor** is a member that implements the actions required to destruct an instance of a class. Destructors are not inherited. Thus, a class has no destructors other than the one that may be declared in it. Since a destructor is required to have no parameters, it cannot be overloaded. Thus, a class can have, at most, one destructor. Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for destruction when it is no longer possible for any code to use the instance.

Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction. When an instance is destructed, the destructors in its inheritance chain are called, in order, from most derived to least derived. Destructors are implemented by overriding the virtual method Finalize on System.Object. Programs are not permitted to override this method or call it (or overrides of it) directly.
8 ARRAYS

8.1 INTRODUCTION

An array is a data structure. It contains a number of variables, which are accessed through computed indices. Also called the elements of the array, the variables contained in an array are all of the same type, which is called the element type of the array.

An array has a rank that determines the number of indices associated with each array element. The rank of an array is also referred to as the dimensions of the array. An array with a rank of one is called a single-dimensional array, while an array with a rank greater than one is called a multi-dimensional array. Multi-dimensional arrays of specific sizes are often referred to by size, as two-dimensional arrays, three-dimensional arrays, and so on.

Each dimension of an array has an associated length that is an integral number greater than or equal to zero. The dimension lengths are not part of the type of the array; instead, they are established when an instance of the array type is created at run-time. The length of a dimension determines the valid range of indices for that dimension. For example, for a dimension of length N, indices can range from 0 to N – 1 inclusive.

The total number of elements in an array is the product of the lengths of each dimension in the array. If one or more of the dimensions of an array have a length of zero, the array is said to be empty. The element type of an array can be any type, including an array type.

8.2 ARRAY TYPES

The System.Array Type

The System.Array type is the abstract base type of all array types. An implicit reference conversion exists from any array type to System.Array, and an explicit reference conversion exists from System.Array to any array type. System.Array itself is not an array-type; rather, it is a class-type from which all array-types are derived. At run-time, a value of type System.Array can be null or a reference to an instance of any array type.

8.3 ARRAY CREATION

Array instances are created by array-creation-expressions or by field or local variable declarations that include an array-initializer. When an array instance is created, the rank and length of each dimension are established and then remain constant for the entire lifetime of the instance. It is not possible to change the rank of an existing array instance, nor is it possible to resize its dimensions. An array instance is always of an array type. The System.Array type is an abstract type that cannot be instantiated. Elements of arrays created by array-creation-expressions are always initialized to their default value.

8.4 ARRAY ELEMENT ACCESS

Array elements are accessed using element-access expressions of the form A[I1, I2, ..., IN], where A is an expression of an array type and each IX is an expression of type int, uint, long, ulong, or of a type that can be implicitly converted to one or more of these types. The result of an array element access is a variable, namely the array element selected by the indices. The elements of an array can be enumerated using a foreach statement.

8.5 ARRAY MEMBERS

Every array type inherits the members declared by the System.Array type.

8.6 ARRAY INITIALIZERS

Array initializers may be specified in field declarations, local variable declarations, and array creation expressions. The context in which an array initializer is used determines the type of the array being initialized. In an array...
creation expression, the array type immediately precedes the initializer. In a field or variable declaration, the array type is the type of the field or variable being declared. When an array initializer is used in a field or variable declaration, such as:

```csharp
int[] ar = {1, 3, 5, 7, 9};
```

it is simply shorthand for an equivalent array creation expression:

```csharp
int[] arr = new int[]{1, 3, 5, 7, 9}
```

For a single-dimensional array, the array initializer must consist of a sequence of expressions that are assignment compatible with the element type of the array. The expressions initialize array elements in increasing order, starting with the element at index zero. The number of expressions in the array initializer determines the length of the array instance being created. For example, the array initializer above creates an `int[]` instance of length 5 and then initializes the instance with the following values:

```csharp
```

For a multi-dimensional array, the array initializer must have as many levels of nesting as there are dimensions in the array. The outermost nesting level corresponds to the leftmost dimension, and the innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. For each nested array initializer, the number of elements must be the same as the other array initializers at the same level. The example:

```csharp
int[,] ar = {{10, 11}, {12, 13}, {14, 15}, {16, 17}};
```

creates a two-dimensional array with a length of four for the leftmost dimension and a length of two for the rightmost dimension:

```csharp
int[,] ar = new int[4, 2];
```

and then initializes the array instance with the following values:

```csharp
ar[0, 0] = 0; b[0, 1] = 11;
ar[1, 0] = 2; b[1, 1] = 13;
ar[2, 0] = 4; b[2, 1] = 15;
ar[3, 0] = 6; b[3, 1] = 17;
ar[4, 0] = 8; b[4, 1] = 19;
```

When an array creation expression includes both explicit dimension lengths and an array initializer, the lengths must be constant expressions and the number of elements at each nesting level must match the corresponding dimension length.
9 Structures

9.1 Definition of a Struct

Structs are similar to classes in that they represent data structures that can contain data members and function members. Unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. The simple types provided by C#, such as int, double, and bool, are in fact all struct types. It is possible to use structs and operator overloading to implement new “primitive” types in the C# language.

9.2 Struct Declaration

A struct-declaration is a type-declaration that declares a new struct.

Struct Modifiers

A struct-declaration may optionally include a sequence of struct modifiers. It is a compile-time error for the same modifier to appear multiple times in a struct declaration. The modifiers of a struct declaration have the same meaning as those of a class declaration.

Struct Interfaces

A struct declaration may include a struct-interfaces specification, in which case the struct is said to implement the given interface types.

Struct Body

The struct-body of a struct defines the members of the struct.

9.3 Members of a Struct

The struct members are the members introduced by its struct-member-declarations and the members inherited from System.ValueType, which inherits from object.

9.4 Class and Struct Differences

Structs are value types. All struct types implicitly inherit from class object. Instance field declarations for a struct are not permitted to include variable initializers. Assignment to a variable of a struct type creates a copy of the value being assigned. The meaning of this is different for structs. A struct is not permitted to declare a destructor.

The default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to null. A struct is not permitted to declare a parameterless instance constructor. Boxing and unboxing operations are used to convert between a struct type and object.

Structs are value types and are said to have value semantics. Classes, on the other hand, are reference types and are said to have reference semantics. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object. With classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other. Because structs are not reference types, it is not possible for values of a struct type to be null.
All `struct` types implicitly inherit from class `object`. A `struct` declaration may specify a list of implemented interfaces, but it is not possible for a `struct` declaration to specify a base class. `Struct` types are never abstract and are always implicitly sealed. The abstract and sealed modifiers are therefore not permitted in a `struct` declaration. Since inheritance is not supported for `structs`, the declared accessibility of a `struct` member cannot be protected or protected internal. Function members in a `struct` cannot be abstract or virtual, and the override modifier is allowed only to override methods inherited from the object type.

Assignment to a variable of a `struct` type creates a *copy* of the value being assigned. This differs from assignment to a variable of a class type, which copies the reference but not the object identified by the reference. Similar to an assignment, when a `struct` is passed as a value parameter or returned as the result of a function member, a copy of the `struct` is created. A `struct` may be passed by reference to a function member using a `ref` or `out` parameter. When a property or indexer of a struct is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs.

A value of a class type can be converted to type `object` or to an interface type that is implemented by the class simply by treating the reference as another type at compile-time. Likewise, a value of type `object` or a value of an interface type can be converted back to a class type without changing the reference (but of course a run-time type check is required in this case). Since `structs` are not reference types, these operations are implemented differently for `struct` types. When a value of a `struct` type is converted to type `object` or to an interface type that is implemented by the `struct`, a boxing operation takes place.

When a value of type `object` or a value of an interface type is converted back to a `struct` type, an unboxing operation takes place. A key difference from the same operations on class types is that boxing and unboxing copies the `struct` value either into or out of the boxed instance. Following a boxing or unboxing operation, changes made to the unboxed `struct` are not reflected in the boxed `struct`.

Within an instance constructor or instance function member of a class, `this` is classified as a value. Thus, while `this` can be used to refer to the instance for which the function member was invoked, it is not possible to assign to `this` in a function member of a class. Within an instance constructor of a `struct`, `this` corresponds to an `out` parameter of the `struct` type, and within an instance function member of a `struct`, `this` corresponds to a `ref` parameter of the `struct` type. `this` is classified as a variable, and it is possible to modify the entire `struct` for which the function member was invoked by assigning to `this` or by passing `this` as a `ref` or `out` parameter.

The default value of a struct consists of the value that results from setting all value type fields to their default value and all reference type fields to `null`. For this reason, a `struct` does not permit instance field declarations to include variable initializers. Unlike a class, a `struct` is not permitted to declare a parameterless instance constructor. Instead, every `struct` implicitly has a parameterless instance constructor that always returns the value that results from setting all value type fields to their default value and all reference type fields to `null`. A `struct` instance constructor is not permitted to include a constructor initializer of the form `base(...)`. The `this` variable of a `struct` instance constructor corresponds to an `out` parameter of the `struct` type, and, similar to an `out` parameter, `this` must be definitely assigned at every location where the instance constructor returns. A `struct` can declare instance constructors having parameters. It is not permitted to declare a destructor.
10 MISCELLANEOUS

10.1 C# 2.0

With the release of Visual Studio 2005, the C# language has been updated to version 2.0.

10.1.1 The “System.Diagnostics” Namespace

The System.Diagnostics namespace provides classes that allow the interaction with system processes, event logs, and performance counters. It also provides classes that allow debugging the application and tracing the code execution.

The “Stopwatch” Class
The Stopwatch class provides a set of methods and properties that can be used to accurately measure elapsed time.

10.1.2 The “System.Drawing” Namespace

The System.Drawing namespace provides access to GDI+ basic graphics functionality.

The “Rectangle” Structure
The Rectangle structure stores a set of four integers that represent the location and size of a rectangle. A rectangle is defined by its width, height, and upper-left corner.

The “Color” Structure
The Color structure represents an “ARGB” color.

The “PointF” Structure
The PointF structure represents an ordered pair of floating point x- and y-coordinates. The pair defines a point in a two-dimensional plane.

The “Point” Structure
The Point structure represents an ordered pair of integer x- and y-coordinates. The pair defines a point in a two-dimensional plane.

The “Font” Class
The Font class defines a particular format for text, including font face, size, and style attributes.

10.1.3 The “System.Windows.Forms” Namespace

The System.Windows.Forms namespace contains classes used for creating Windows-based applications that take full advantage of the user interface features available in the Microsoft Windows operating system.

The “Form” Class
The Form class represents a window or dialog box that makes up an application’s user interface.

10.1.4 The “System.Collections.Generic” Namespace

The System.Collections.Generic namespace contains interfaces and classes that define generic collections. These interfaces and classes allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.
The “List<T>” Class

The List<T> class implements the IList<T> interface using an array whose size is dynamically increased as required.

10.2 Generics

Generics are a new feature in version 2.0 of the C# language. Generic types are added to the language to enable the programmer to achieve a high level of code reuse and enhanced performance for collection classes. They are used with collections and the methods that operate on them. Generics introduce the concept of type parameters. Generic classes encapsulate operations that are not specific to any particular data type. The most common use for generic classes is with collections like:

1. Linked lists
2. Hash tables
3. Stacks
4. Queues
5. Trees

It is useful to define interfaces either for generic collection classes, or for the generic classes that represent items in the collection. It is preferable to use generic interfaces with generic classes.

A generic method is a method that is declared with type parameters. Non-generic methods can access the class-level type parameters within a generic class. Generic methods can be overloaded on a number of type parameters. Delegates defined within a generic class can use the generic class type parameters in the same way that class methods do. Generic delegates are especially useful in defining events based on the typical design pattern. A new namespace called System.Collections.Generic includes several ready-to-use generic collection classes and associated interfaces.

10.3 The “Exception” Class

The Exception class represents errors that occur during application execution. This class is the base class for all exceptions. When an error occurs, either the system or the currently executing application reports it by throwing an exception containing information about the error. Once thrown, an exception is handled by the application or by the default exception handler.

If an application handles exceptions that occur during the execution of a block of application code, the code must be placed within a try statement. When an exception occurs in a try block, the system searches the associated catch blocks in the order they appear in application code, until it locates a catch block that handles the exception. A catch block handles an exception of type T if the type filter of the catch block specifies T or any type that T derives from. The system stops searching after it finds the first catch block that handles the exception.

10.4 DirectX

DirectX is a set of interfaces for creating games and other high-performance multimedia applications. It supports two-dimensional (2-D) and three-dimensional (3-D) graphics, sound effects and music, input devices, and networked applications.

10.4.1 Direct3D

Direct3D enables you to manipulate visual models of 3-dimensional objects and take advantage of hardware acceleration, such as video graphics cards.

The “RenderStateManager” Class

The RenderStateManager class defines device render states.

The “Texture” Class
The Texture class manipulates a texture resource.

The “PresentParameters” Class
The PresentParameters class describes the presentation parameters.

The “Device” Class
The Device class performs primitive-based rendering, creates resources, handles system-level variables, adjusts gamma ramp levels, gets and sets palettes, and creates shaders.

The “Sprite” Class
The Sprite class provides methods and properties that simplify the process of drawing sprites using Direct3D.

The “Font” Class
The Font class encapsulates the textures and resources needed to render a specific font on a specific device.

10.4.2 DirectInput
DirectInput is used to process data from a keyboard, mouse, joystick, or other game controller.

The “Device” Class
The Device class is used to gain and release access to DirectInput devices, manage device properties and information, set behavior, perform initialization, create and play force-feedback effects, and invoke a device’s control panel.

The “Key” Enumeration
The Key enumeration includes all the available keyboard keys.

10.4.3 DirectSound
DirectSound is used to capture sounds from input devices and play sounds through various playback devices using advanced 3-dimensional positioning effects and filters for echo, distortion, reverberation, and other effects.

The “Device” Class
The Device class contains methods and properties that are used to create buffer objects, manage devices, and set up the environment.

The “SecondaryBuffer” Class
The SecondaryBuffer class contains methods and properties that are used to manage sound buffers that can support effects.

10.4.4 DirectX.AudioVideoPlayback
The AudioVideoPlayback interface provides for basic playback and simple control of audio and video files.

The “Audio” Class
The Audio class is primarily designed for very simple playback scenarios, or for use with the Video class.

10.4.5 The “Vector2” Class
The Vector2 class describes and manipulates a vector in two-dimensional space.
10.4.6 The “Matrix” Class

The Matrix class describes and manipulates a matrix.