Object Oriented Programming with C#

MD. MAHEDEE HASAN

Object Oriented Programming with C#

5th Edition

Md. Mahedee Hasan
Microsoft Most Valuable Professional (MVP)
Visual Studio and Development Technologies
Software Architect
Leadsoft Bangladesh Limited
Blog: ***http://mahedee.net/***
Email: ***mahedee.hasan@gmail.com***
Linkedin: ***http://www.linkedin.com/in/mahedee***
Facebook: ***https://www.facebook.com/mahedee.net***

Chapter 1: Introduction to Visual Studio

What is Microsoft Visual Studio?

- Microsoft Visual Studio is a **wonderful integrated development environment (IDE)** from Microsoft.
- It's **user friendly** and **smart**.
- It is used to **develop computer program** for Microsoft windows.
- It is also used to develop **website, web application, web service and web API**.
- Visual studio code editor supports **intelliSence** and **code refactoring.**
- It is very smart in the world of **IDE**.
- It supports many languages such as:
    - C, C++, C#, VB.NET, F#.
    - It also supports Phython and Ruby.
    - Support HTML/XHTML, CSS, JavaScript, XML/XSLT

Different version of Visual Studio

Product name	Codename	Internal version	Supported .NET Framework versions	Release date
Visual Studio	N/A	4.0	N/A	April 1995
Visual Studio 97	Boston	5.0	N/A	February 1997
Visual Studio 6.0	Aspen	6.0	N/A	June 1998
Visual Studio .NET (2002)	Rainier	7.0	1.0	February 13, 2002
Visual Studio .NET 2003	Everett	7.1	1.1	April 24, 2003
Visual Studio 2005	Whidbey	8.0	2.0, 3.0	November 7, 2005
Visual Studio 2008	Orcas	9.0	2.0, 3.0, 3.5	November 19, 2007
Visual Studio 2010	Dev10/Rosario	10.0	2.0, 3.0, 3.5, 4.0	April 12, 2010
Visual Studio 2012	Dev11	11.0	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2	September 12, 2012
Visual Studio 2013	Dev12	12.0	2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2	October 17, 2013

Visual Studio 2015	Dev14	14.0	2.0 – 4.6	July 20, 2015

Chapter 2: C# Fundamentals

An Introduction to C#

- C# is intended to be a **simple**, **modern**, **general-purpose, object-oriented** programming language
- It is very **sophisticated programming** language
- It is developed by Microsoft with its .NET initiatives
- The language is **intended** for use in developing **Software Components**

C# Program Structure

- Program execution begins at **Main()**
- The **using keyword** refers to resources in the .NET framework class library
- **Statements are commands** that perform actions
    - A program is made up of **many separate statement**
    - Statement are **separated by a semicolon**
    - Braces are used to **group statement**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MyFirstProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Welcome to MAHEDEE.NET");
            Console.ReadKey();
        }
    }
}
```

C# Code Formatting

- Use **indentation** to indicate enclosing statements
    - Indentation indicates that a statement is within an enclosing statement. Statements that are in the **same block of statements should all be indented to the same level**. This is an **important convention** that improves the **readability** of your code.
- C# is **case sensitive**

- **White space** is ignored
- Indicate single line comment by using //
- Indicate multiple line comment by using /*and*/

Example:

```
class Formatting
{

    static void Main(string[] args)
    {
        int value = 10;
        int VALUE = 20;

        Console.WriteLine("Small value: " + value);
        Console.WriteLine("Big value: " + VALUE);
        Console.WriteLine("White     space is "+                "ignored by C#");

        //This is a single line comment

        /*
        This is multiline comment.
        Use this for commenting multiple lines in C#.
        */

        Console.ReadKey();
    }
}
```

Chapter 3: C# Data Types

Predefined types

- **Types** are used to **declare variables**
- **Variables store** different kinds of **data**
    - Let the **data** that you are representing determine **your choice of variable**
- Predefined types are those **provided by C# and .NET framework**
    - You can also defined your own
- Variable must be **declared before you can use them**
- Whenever your application must store data temporarily for use during execution, you **store that data in a *variable***.
- You can think of **variables as storage boxes**. These **boxes come in different sizes and shapes, called *types***, which provide storage for various kinds of data.

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

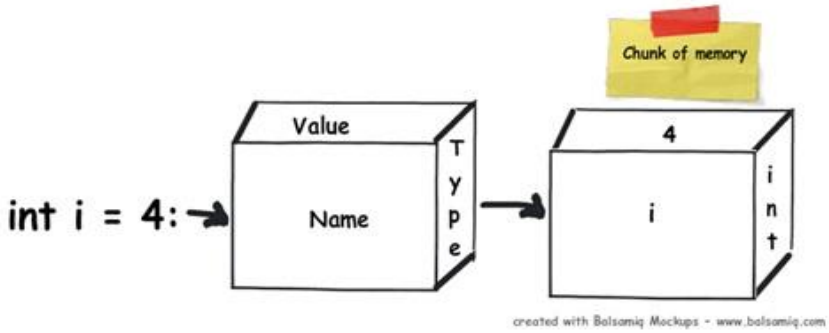
- **Value types**
- **Reference types**

Value Types

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

What goes inside when you declare a variable?

- When you **declare a variable** in a .NET application, it allocates **some chunk of memory** in the **RAM**.
- This memory has **three things: the name of the variable, the data type of the variable, and the value of the variable.**
- That was a simple explanation of what happens in the memory, but depending on the data type, your variable is allocated that type of memory.
- There are **two types of memory allocation: stack memory and heap memory.**



Stack and heap

In order to understand stack and heap, let's understand what actually happens in the below code internally.

```
public void Method1()
{
    // Line 1
    int i = 4;

    // Line 2
```

```
            int y = 2;

            //Line 3
            class1 cls1 = new class1();

        }
```

It's a three line code, let's understand line by line how things execute internally.

- **Line 1**: **When this line is executed, the compiler allocates a small amount of memory in the stack**. The stack is responsible for keeping track of the running memory needed in your application.

- **Line 2**: Now the execution moves to the **next step**. As the name says stack, it stacks this memory allocation on **top of the first memory allocation**. You can think about stack as a series of compartments or boxes put on top of each other.

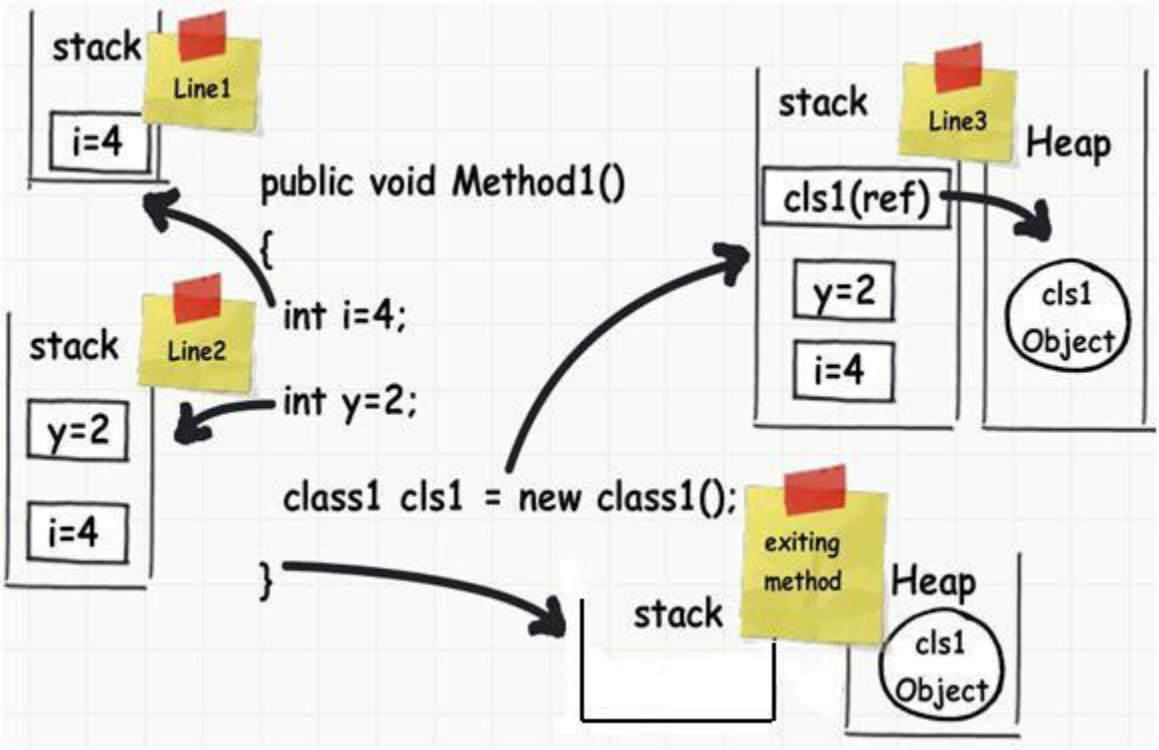
  Memory allocation and de-allocation is done using LIFO (Last In First Out) logic. In other words memory is allocated and de-allocated at only one end of the memory, i.e., top of the stack.

- **Line 3**: In line 3, we have created an **object**. When this line is executed it **creates a pointer on the stack** and the **actual object is stored in a different type of memory location called 'Heap'**. 'Heap' does not track running memory, it's just a pile of objects which can be reached at any moment of time. Heap is used for **dynamic memory allocation.**

  One more important point to note here is reference pointers are allocated on stack. The statement, Class1 cls1; does not allocate memory for an instance of Class1, it only allocates a stack variable cls1 (and sets it to null). The time it hits the new keyword, it allocates on "heap".

  Exiting the method (the fun): Now finally the execution control starts exiting the method. **When it passes the end control, it clears all the memory variables** which are assigned on stack. In other words all variables which are related to `int` data type are de-allocated in 'LIFO' fashion from the stack.

  The **big** catch – It did not de-allocate the heap memory. This memory will be later de-allocated by the garbage collector.



The following table lists the available value types in C#:

Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	(-7.9 x 1028 to 7.9 x 1028) / 100 to 28	0.0M
double	64-bit double-precision floating point type	(+/-)5.0 x 10-324 to (+/-)1.7 x 10308	0.0D
float	32-bit single-precision floating point type	-3.4 x 1038 to + 3.4 x 1038	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-923,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0

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

Sample Code:

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

How to declare and initialize variable

- A variable is a **storage location** for a particular type
- **Declaring**
  - Assign a **type**
  - Assign a **name**
  - End with a **semicolon**
  - Ex. int noOfUser; string firstName;
- **Initializing**
  - Use **assignment operator**
  - Assign a **value**
  - End with a **semicolon**
  - Ex. string firstName = "Mahedee";
- Assigning **literal variable**
  - Add a type **suffix**
  - Ex. decimal deposit = 50000M;

The following list identifies some **best practices for naming your variables**:

- Assign **meaningful names** to your variables.
- Use **camel case**. In camel case, the first letter of the **identifier** is lowercase, and the first letter of each subsequent word in the identifier is capitalized, such as **newAccountBalance.**
- Do not use C# **keywords.**
- Although C# is case sensitive, **do not create variables that differ only by case**.

Example:

```
class Program
{
    static void Main(string[] args)
    {
        int myVariable;
        myVariable = 1;

        int x = 25;
        int y = 50;
        bool isOpen = false;
        sbyte b = -55;

        Console.WriteLine(myVariable);
        Console.WriteLine(x);
        Console.ReadKey();
    }
}
```

Literal

A literal is a **value** that has been **hard-coded** directly into your **source.**

For example:

```
string ltrlString = "This is a literal";
int y = 2; // so is 2, but not y
int z = y + 4; // y and z are not literals, but 4 is
int a = 1 + 2; // 1 + 2 is not a literal (it is an expression), but 1 and 2 considered
```

Assigning literal values

When you assign 25 to x in the preceding code, the compiler places the literal value 25 in the variable x. The following assignment, however, generates a compilation error:

```
decimal bankBalance = 3433.20;   // ERROR!
```

Error: Literal of type **double cannot be implicitly converted to type 'decimal'**; use an 'M' suffix to create a literal of this type

This code causes an error because the **C# compiler assumes than any literal number with a decimal point is a *double***, unless otherwise specified. You specify the type of the literal by appending a suffix, as shown in the following example:

```
decimal bankBalance = 3433.20M;
```

The literal suffixes that you can use are shown in the following table. **Lowercase is permitted.**

Category	Suffix	Description
Integer	U	Unsigned
	L	Long
	UL	Unsigned long
Real number	F	Float
	D	Double
	M	Decimal
	L	Long

Characters

You specify a character (char type) by enclosing it in single quotation marks:

```
char myInitial = 'a';
```

Escape characters

Some characters cannot be specified by being placed **in quotation marks**—for example, a **newline character, a beep, or a quotation mark character**. To represent these characters, you must use **escape characters**, which are shown in the following table.

Escape sequence	Character name
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash
\0	Null
\a	Alert
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

For example, you can specify a quotation mark as follows:

```
char quoteMark = '\'';
```

Initialize String

- Example string
  - `string str = "Hello world"; //Hello world`
- Literal string
  - `string str = "\"Hello\""; //"Hello"`
- Escape character
  - `string str = "Hello\nWorld"; // a new line is added between Hello and World`
- Using **verbatim** string
  - `string str = @"Hello\n"; //Hello\n`
- Understanding Unicode
  - The character "A" is represented by "U+0041"
  - Example:

```
char c = '\u0041';
Console.WriteLine(c.ToString());
```

To insert a backslash, which is useful for including file path locations, use the \\ escape character.

Verbatim String

- **A *verbatim* string is a string** that is **interpreted** by the **compiler** exactly **as it is written**
  - Which means that even if the string spans **multiple lines** or includes **escape characters**, these are not interpreted by the compiler and they are included with the output.
  - The only **exception is the quotation mark** character, which must be escaped so that the compiler can recognize where the string ends.
- A verbatim string is indicated with an at **sign (@)** character followed **by the string enclosed in quotation marks.** For example:

```
string sample = @"Hello";
string sample = @"Hello\tWorld"; // produces Hello\tWorld
```

  - If you want to **use a quotation mark inside a verbatim string**, you must **escape it by using another set of quotation marks**. For example, to produce "Hi" you use the following code:

```
string s = @"""Hi"""; // Note: three quotes on either side
```

The preceding code produces the following string:

"Hi"

Understanding Unicode

- **The .NET Framework uses Unicode UTF-16 (Unicode Transformation Format, 16-bit encoding form) to represent characters.**
- C# also encodes characters by using the international Unicode Standard.
- The Unicode Standard is the **current universal character encoding mechanism** that is used to represent text in computer processing. **The previous standard was ASCII.**
- The Unicode Standard represents a significant improvement over ASCII because Unicode assigns a **unique numeric value, called a code point**, and a name to each character that is used in all the written languages of the world.
- **ASCII defined only 128 characters**, which meant that some languages could not be correctly displayed in a computer application.

For example, the character **"A" is represented by the code point "U+0041"** and the name "LATIN CAPITAL LETTER A". Values are available for over **65,000 characters**, and there is room to support up to one million more. For more information, see The Unicode Standard at **www.unicode.org.**

Sample Code:

```
class Program
{
    static void Main(string[] args)
    {

        string str1 = "Welcome to MAHEDEE.NET"; //Welcome to MAHEDEE.NET
        string str2 = "\"Hello\"";
        string str3 = "Trainee,\nWelcome to MAHEDEE.NET. once again";
        string str4 = @"\tThis is the varbatim sign\n";
        Console.WriteLine(str1);
        Console.WriteLine(str2);
        Console.WriteLine(str3);
        Console.WriteLine(str4);

        string sample = "c:\\My Documents\\sample.txt";
        Console.WriteLine(sample);

        string sample2 = @"c:\My Documents\sample.txt";
        Console.WriteLine(sample2);


        string sample3 = @""""";
        Console.WriteLine(sample3);


        string sample4 = @"""Hi";
        Console.WriteLine(sample4);

        string sample5 = @"""Hi""";
        Console.WriteLine(sample5);


        Console.WriteLine("\uFEFF");
        Console.WriteLine("\u0041");
```

```
            Console.ReadKey();

        }
    }
```

Output:

Welcome to Leads Technology
"Hello"
Trainee,
Welcome to Leads Technology Ltd. once again
\tThis is the varbatim sign\n
c:\My Documents\sample.txt
c:\My Documents\sample.txt
"
"Hi
"Hi"
?
A

Convert between type

- **Implicit**
    - Performed by the compiler on operations that are guaranteed **not to truncate information**

```
int x = 123456; // int is a 4-byte integer
long y = x; // implicit conversion to a long
```

- **Explicit**
    - Where you to **explicitly ask the compiler** to perform a conversion that otherwise could **lose information**

```
int x = 500;
short z = (short)x; //explicit conversion to a short, z contains the value
500; short is 2 byte data type
```

Implicit conversions

The following table shows the implicit type conversions that are supported in C#:

From	To
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal

int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

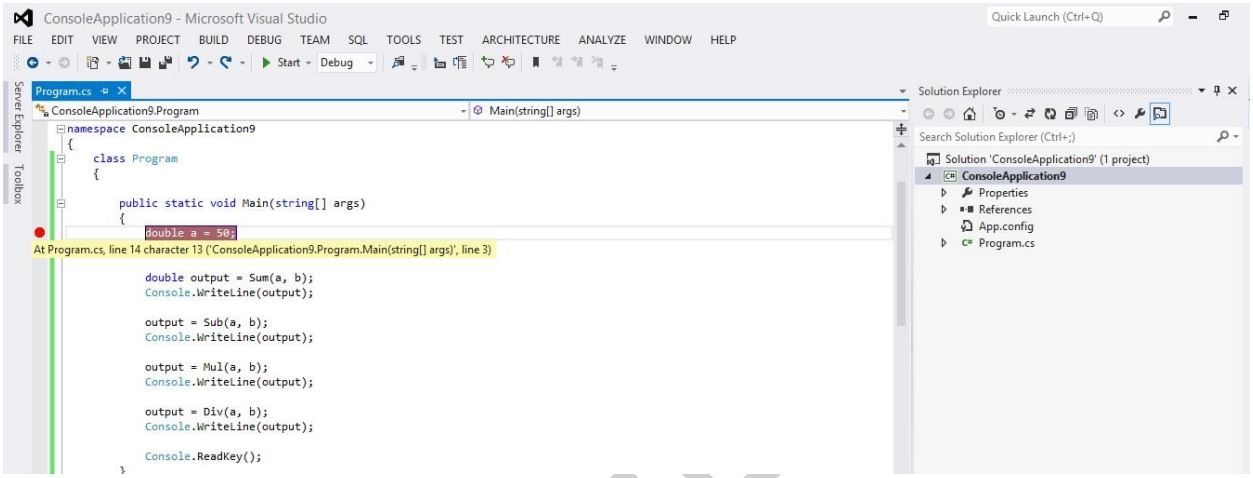
Constant

- Declare **using const keyword** and type
- You **must assign** a value at the time of declaration
- Examples
  - `const double pi = 3.14;`
  - `const int earthRadius = 6378;`

```
class Program
{
    const int earthRadius = 6378; // km
    const long meanDistanceToSun = 149600000; // km
    const double meanOrbitalVelocity = 29.79D; // km/sec


    static void Main(string[] args)
    {
        Console.WriteLine("Earth Radius: " + earthRadius);
        Console.WriteLine("Mean distance to Sun" + meanDistanceToSun);
        Console.WriteLine("Mean Orbital Velocity" + meanOrbitalVelocity);

        //earthRadius = 5000; //Error

        Console.ReadKey();
    }
}
```

Chapter 4: How to debug Application using Visual Studio

Setting break point



Debugging: Stepping Through Code

- New Break point = SHIFT+B = F9
- Toggle Break point = F9

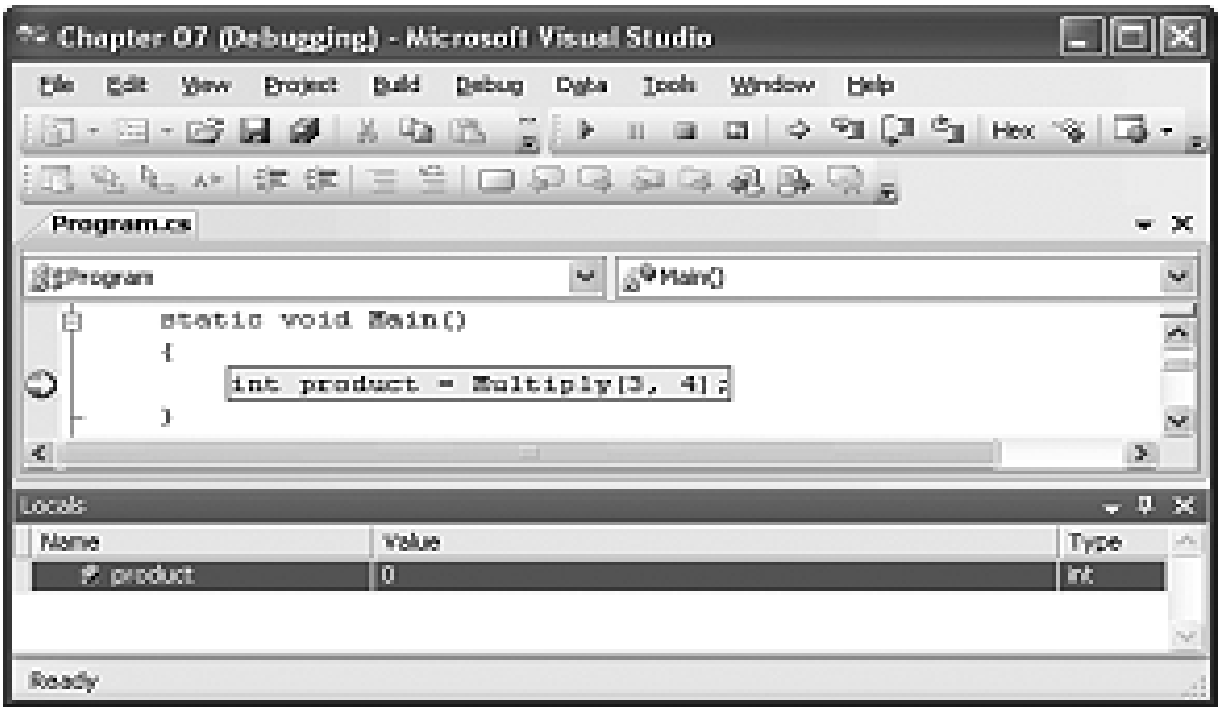
- Step In = **F11** = Step into a Method
- Step Out = Shift + F11 = Steps up and out of a method back to the caller

- Step Over = **F10** = Steps past a method to the next statement

- Start debugging = F5
- Stop Debugging = Shift + F5 = Stops a debugging session

- Start without debugging = Ctrl + F5

Debugging: Examining Program State



Example:

```
class Program
{

    public static void Main(string[] args)
    {
        double a = 50;
        double b = 5;

        double output = Sum(a, b);
        Console.WriteLine(output);

        output = Sub(a, b);
        Console.WriteLine(output);

        output = Mul(a, b);
        Console.WriteLine(output);

        output = Div(a, b);
        Console.WriteLine(output);

        Console.ReadKey();
    }

    public static double Sum(double a, double b)
    {
```

```
        return a + b;
    }

    public static double Sub(double a, double b)
    {
        return a - b;
    }

    public static double Mul(double a, double b)
    {
        return a * b;
    }

    public static double Div(double a, double b)
    {
        return a / b;
    }


}
```

Chapter 5: Enumeration Type

What is Enumeration Type

- **User defined** data type
- **Purpose** of enumeration type is to use **constant values**
- **Process** to create enumeration type
    - **Create** an enumeration type
    - **Declare** variable of that type
    - **Assign** values to those variables

Defining enumeration types

- Create an enumeration type

```
enum BluechipTeam
{
    Azad,
    Mahedee,
    Hasan,
    Sinthi
}
```

- Using enumeration types

```
BluechipTeam aMember = BluechipTeam.Mahedee;
```

- Displaying the variables

```
Console.WriteLine(aMember);
```

Sample Code

```
enum BluechipTeam
{
    Azad,
    Mahedee,
    Sarwar,
    Jamil
}

class Program
{
    static void Main(string[] args)
    {
        BluechipTeam aMember = BluechipTeam.Mahedee;
        Console.WriteLine(aMember);
        Console.ReadKey();
    }
}
```

Assigning values to enumeration members

```
enum Planets
{
    Mercury = 2437,
    Venus = 6095,
    Earth = 6378
}
```

Enumeration base types

```
enum Planets : uint
{
    Mercury = 2437,
    Venus = 6095,
    Earth = 6378
}
```

Sample Code:

```
enum BluechipTeam
{
    Azad,
    Mahedee,
    Sarwar,
    Jamil
}

enum Planets
{
    Mercury = 2437,
```

```
        Venus = 6095,
        Earth = 6378
    }


    class Program
    {
        static void Main(string[] args)
        {
            BluechipTeam aMember = BluechipTeam.Mahedee;
            Console.WriteLine(aMember);

            Planets aPlanet = Planets.Earth;
            Console.WriteLine(aPlanet);
            Console.WriteLine(Convert.ToInt32(aPlanet));
            Console.ReadKey();
        }
    }
```

Chapter 6: Expressions & Operators

What is Expression?

- An expression is a **sequence of operators and operands**
- The **purpose** of writing an expression is to **perform an action and return a value**
- Operators are symbols used in expressions

Common Operators	Example
Increment / decrement	++ --
Arithmetic	* / % + -
Relational	< > <= >=
Equality	== !=
Conditional	&& || ?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

Types of Operators

The following table lists all the operators that can be used in a C# application:

Operator type	Operator
Primary	(x), x.y, f(x), a[x], x++, x--, new, typeof, sizeof, checked, unchecked
Unary	+, -, !, ~, ++x, --x, (T)x
Mathematical	+ , - ,*, /, %

Shift	<< , >>
Relational	< , > , <= , >= , is
Equality	= =
Logical	& , | , ^
Conditional	&& , || , ?
Assignment	= , *= , /= , %= , += , -= , <<=, >>= , &= , ^= , |=

Increment and decrement

- The increment and decrement operators can **occur** either **before or after an operand**.
- For example, x++ and ++x are both **equivalent to x=x+1**.
- However, when these operators **occur** in expressions, x++ and ++x **behave differently**.

++x increments the value of x *before* the expression is evaluated. In other words, x is incremented and then the new value of x is used in the expression.

Example:

```
int x = 5;
(++x == 6) // true or false?
```

The answer is **true**.

x++ increments the value of x after the expression is carried out; therefore, the expression is evaluated using the original value of x.

Example:

```
x = 5
(x++ == 6) // true or false?
```

The answer is **false**.

Example:

```
int x = 10
int y = x++; // y is equal to ten
int z = x + y; // z is equal to twenty-one
```

Example:

```
class Program
{
    static void Main(string[] args)
    {
        int aValue = 5;

        if (++aValue == 6)
        {
```

```
                Console.WriteLine("Increments the value before the expression is
evaluated.");
            }

            if (aValue++ == 6)
            {
                Console.WriteLine("Increments the value after the expression is
evaluated.");
            }

        }
    }
```

Mathematical operators

- Some mathematical operator +, -, *, /
- **Remainder** operator (%) is a mathematical operator, returns the remainder of a division operation

For example:

```
int x = 20 % 7;    // x == 6
```

Logic Operators

C# provides logic operators, as shown in the following table.

Logic operator type	Operator	Description
Conditional	&&	x && y returns true if x is true AND y is true; y is evaluated only if x is true
	||	x || y returns true if x is true OR y is true; y is evaluated only if x is false
Boolean	&	x & y returns true if x AND y are both true
	|	x | y returns true if either x OR y is true
	^	x ^ y returns **true if x OR y is true**, **but false if they are both true or both false**

Primary Operators

Expression	Description
x.y	Member access

f(x)	Method and delegate invocation
a[x]	Array and indexer access
x++	Post-increment
x--	Post-decrement
new T(...)	Object and delegate creation
new T(...){...}	Object creation with initializer.
new {...}	Anonymous object initializer.
new T[...]	Array creation.
typeof(T)	Obtain System.Type object for T
checked(x)	Evaluate expression in checked context
unchecked(x)	Evaluate expression in unchecked context
default (T)	Obtain default value of type T
delegate {}	Anonymous function (anonymous method)

Unary Operators

Expression	Description
+x	Identity
-x	Negation
!x	Logical negation
~x	Bitwise negation
++x	Pre-increment
--x	Pre-decrement
(T)x	Explicitly convert x to type T

Multiplicative Operators

Expression	Description
*	Multiplication
/	Division
%	Remainder

Additive Operators

Expression	Description
x + y	Addition, string concatenation, delegate combination
x - y	Subtraction, delegate removal

Shift Operators

Expression	Description
x << y	Shift left
x >> y	Shift right

Relational and Type Operators

Expression	Description
x < y	Less than
x > y	Greater than
x <= y	Less than or equal
x >= y	Greater than or equal

x is T	Return true if x is a T, false otherwise
x as T	Return x typed as T, or null if x is not a T

Equality Operators

Expression	Description
x == y	Equal
x != y	Not equal

Logical, Conditional, and Null Operators

Category	Expression	Description
Logical AND	x & y	Integer bitwise AND, Boolean logical AND
Logical XOR	x ^ y	Integer bitwise XOR, boolean logical XOR
Logical OR	x | y	Integer bitwise OR, boolean logical OR
Conditional AND	x && y	Evaluates y only if x is true
Conditional OR	x || y	Evaluates y only if x is false
Null coalescing	x ?? y	Evaluates to y if x is null, to x otherwise
Conditional	x ?: y : z	**Evaluates to y if x is true, z if x is false**

Assignment and Anonymous Operators

Expression	Description
=	Assignment
x op= y	Compound assignment. Supports these operators: +=, -=, *=, /=, %=, &=, |=, !=, <<=, >>=

(T x) => y	Anonymous function (lambda expression)

References: http://msdn.microsoft.com/en-us/library/ms173145.aspx

Using operators with strings

- You can also apply **the plus and the equality operators** to string types.
- The **plus concatenates strings** whereas the string equality operator compares strings.

```
string a = "semi";
string b = "circle";
string c = a + b;
string d = "square";
```

The string c has the value **semicircle**.

```
bool sameShape = ("circle" == "square");

sameShape = (b == d);
```

The Boolean **sameShape** is **false** in both statements.

Operator Precedence

- Expressions are evaluated according to **operator precedence**
    - Example: 10 + 20 / 5 (result is 14)
- **Parenthesis** can be used to **control the order** of evaluation.
    - Ex. (10 + 20) / 5 (result is 6)
- Operator **precedence** is also determined by **associativity**
    - **Binary operators** are **left** associative i.e evaluated from left to right.
    - **Assignment** and **conditional** operators are right associative i.**e evaluated from right to left**

Evaluation order

The order in which operators are evaluated in an expression is shown in the following precedence table.

Operator type	Operator
Primary	x.y, f(x), a[x], x++, x--, new, typeof, checked, unchecked
Unary	+, -, !, ~, ++x, --x, (T)x
Multiplicative	*, /, %
Additive	+ , -
Shift	<< , >>
Relational	< , > , <= , >= , is, as
Equality	== , !=
Logical	& , ^ , |
Conditional	&& , || , ?:
Assignment	= , *= , /= , %= , += , -= , <<=, >>= , &= , ^= , |=

For example, **the plus operator + has a lower precedence than the multiplication operator**, so a + b * c means multiply b and c, and then add the sum to a.

Parentheses

- Use parentheses to show the **order of evaluation** and to make the evaluation order of your expressions more readable.
- **Extra parentheses** are **removed by the compiler** and do not slow your application in any way, but they can make an expression much more readable.

For example, in the following expression, the compiler will multiply b by c and then add d.

```
a = b * c + d
```

Using parentheses, in the following expression, the compiler first evaluates what is in parentheses, (c + d), and then multiplies by b.

```
a = b * (c + d)
```

The following examples demonstrate operator precedence and the use of parentheses for controlling the order of evaluation in an expression:

```
10 + 20 / 5  (result is 14)
(10 + 20) / 5  (result is 6)
10 + ( 20 / 5 )  (result is 14)
((10 + 20) * 5) + 2  (result is 152)
```

Chapter 7: Control or Conditional Statement

- A conditional statement allows you to **control the flow** of your application
  - By selecting the statement that is **executed, based on the value of a Boolean expression**

Use if statement

- ***if*** statement

```
if ( sales > 10000 ) {
        bonus += .05 * sales;
}
```

- ***if else*** statement

```
if ( sales > 10000 ) {
        bonus += .05 * sales;
}
else {
        bonus = 0;
}
```

- ***if else if*** statement

```
if ( sales > 10000 ) {
        bonus += .05 * sales;
}
else if ( sales > 5000 ) {
        bonus = .01 * sales;
}
else    {
        bonus = 0;
        if ( priorBonus == 0 ) {
        // Schedule a Meeting;
        }
}
```

Example:

```
class Program
{
    static void Main(string[] args)
    {
        int sales = 0;
        double bonus = 0;
        string input = string.Empty;

        input = Console.ReadLine();
        bool isParsed = Int32.TryParse(input, out sales); //not using Convert
```

```
            if (!isParsed)
                sales = 0;

            //sales = Convert.ToInt32(Console.ReadLine());

            if (sales > 10000)
            {
                bonus += .05 * sales;
            }
            else if (sales > 5000)
            {
                bonus += .01 * sales;
            }
            else
            {
                bonus = 0;
                Console.WriteLine("Sorry, you could not achieve sales target!");
            }

            Console.WriteLine("Your bonus is: " + bonus);
            Console.ReadKey();
        }
    }
```

Switch Statement

- **Switch** statements are useful for **selecting one branch of execution from a list of mutually-exclusive choices.**

```
            switch (favoriteAnimal)
            {
                case Animal.Antelope:
                    // herbivore-specific statements
                    break;
                case Animal.Elephant:
                    // herbivore-specific statements
                    break;
                case Animal.Lion:
                    // carnivore-specific statements
                    break;
                case Animal.Osprey:
                    // carnivore-specific statements
                    break;
                default:
                    //default statemennt
                    break;
            }
```

Example:

```
class Program
{
    enum CapitalMarketTeam
    {
        FaizurRahman,
```

```
            TonmoyShaha,
            Azad,
            Mahedee,
            Enamul,
            Ehsan
        }

        static void Main(string[] args)
        {
            CapitalMarketTeam aMember = CapitalMarketTeam.FaizurRahman;
            //CapitalMarketTeam aMember = CapitalMarketTeam.Ety;

            switch (aMember)
            {
                case CapitalMarketTeam.FaizurRahman:
                    Console.WriteLine("Faizur Rahman, Manager of Capital Market Team");
                    break;

                case CapitalMarketTeam.Azad:
                    Console.WriteLine("Saifullah Al Azad, Software Architect of Capital Market
Team");
                    break;

                case CapitalMarketTeam.TonmoyShaha:
                    Console.WriteLine("Tonmoy Shaha, Business Analyst of Capital Market
Team");
                    break;

                case CapitalMarketTeam.Mahedee:
                    Console.WriteLine("Md. Mahedee Hasan, Software Architect of Capital Market
Team");
                    break;

                case CapitalMarketTeam.Ehsan:
                    Console.WriteLine("Ehsanur Rahman, Software Engineer of Capital Market
Team");
                    break;

                default:
                    Console.WriteLine("Capital Market Team consists of Bluechip, MBank and
Capita Team");
                    break;
            }

            Console.ReadKey();
        }
    }
```

Iteration Statements

- C# provides several looping mechanisms
    - which enable you to **execute a block of code repeatedly** until a certain condition is met
- Looping mechanism
    - **for** loop.

- **while** loop.
- **do** loop.

for loop

- Use when you **know how many times** you want to **repeat** the execution of the code
- Syntax:

for (initializer; condition; iterator) {
    statement-block
}

- Example:

```
for ( int i = 0; i < 10; i++ ) {
       Console.WriteLine( "i = {0}",i );
}
```

```
class Program
{
    static void Main(string[] args)
    {
        int n = 50;
        int sum = 0;

        for (int i = 1; i <= n; i++)
        {
            Console.WriteLine("Sum of {0} to {1} = {2}", 1, i, sum=sum+i);
        }

        Console.ReadKey();
    }
}
```

while loop

- A Boolean test runs at the start of the loop and if tests as False, the loop is never executed.
- The loop **executed until the condition becomes false.**
- Syntax:

while (true-condition) {
    statement-block
}

Example

```
        while (i <= 10)
        {
```

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

Example:

```
class Program
{
    static void Main(string[] args)
    {
        int n = 50;
        int i = 1;
        while (i <= n)
        {
            Console.Write(i + " ");
            i += 2;
        }
        Console.ReadKey();
    }
}
```

Continue, Break

- **The continue** keyword to start the **next loop iteration without executing any remaining statements**
- The **break** keyword is encountered, the **loop is terminated**
- Example:

```
class Program
{
    static void Main(string[] args)
    {
        int i = 0;
        while (true)
        {
            i++;
            if (i > 5 && i <= 7)
                continue;
            if (i >= 10)
                break;
            Console.WriteLine(i);
        }

        Console.ReadKey();

    }
}
```

do loop

- Executes the code in the loop and **then performs a Boolean test**. If the expression as true then the loop repeats until the expression test as false

- Syntax:

```
do {
    statements
} while (boolean-expression);
```

- Example:

```
class Program
{
    static void Main(string[] args)
    {
        int i = 1;
        do
        {
            Console.WriteLine("{0}", i++);
        } while (i <= 10);

        Console.ReadKey();
    }
}
```


Chapter 8: Introduction to Object Oriented Programming

What Procedural Programming?

- These languages code programs in such a way that the program executes statement by statement, reading and modifying a shared memory.
- This programming style can be closely associated with the conventional sequential processors linked to a random access memory (RAM).

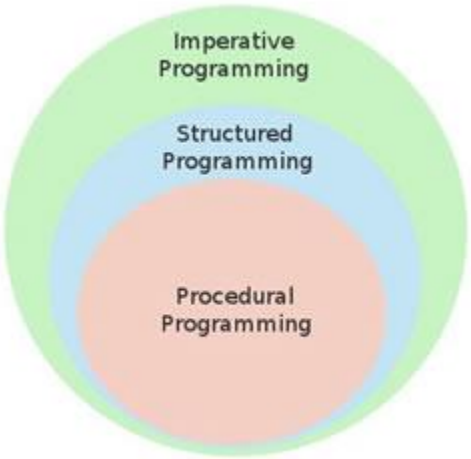
What is Structured Programming?

- These are based on the top down methodology in which a system is further divided into compositional subsystem.

What is Object Oriented Programming?

- Object-oriented programming (OOP) is a **programming paradigm**
- U**ses** "**Objects** and their **interactions** to **design applications** and computer programs".
- Based on **the concepts of classes and objects** that are used for **modeling the real world entities**
- Consist of a group **of cooperating objects**
- **Objects exchange messages**, for the purpose of achieving a common objective

- Implemented in object-oriented languages



Benefit of OOP

- Programs are **easier to design** because objects reflect **real-world items.**
- Applications are easier for users because data they **do not need is hidden.**
- Objects are **self-contained units**.
- **Productivity increases** because you can **reuse code**.
- Systems **are easier to maintain** and adapt to changing business needs.

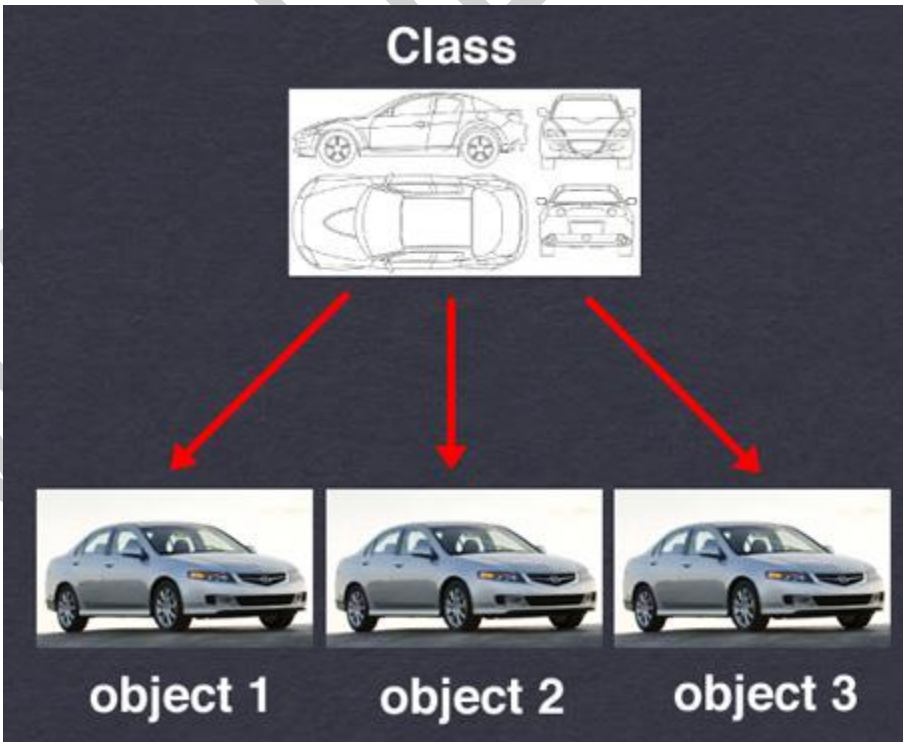
Difference between Structured and Object Oriented Programming

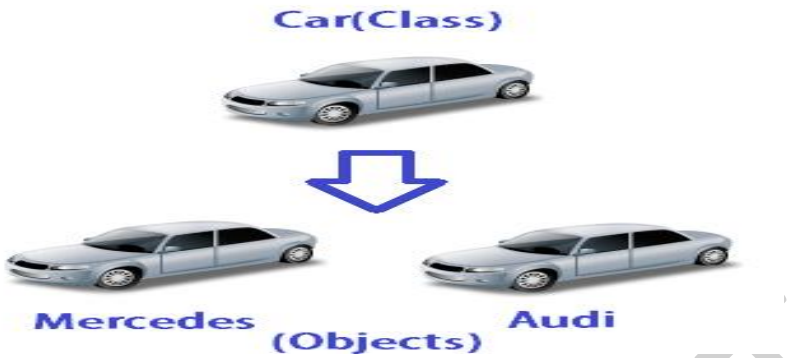
Structured Programming	Object Oriented Programming
Structured Programming is designed which focuses on **process/ logical** structure and then data required for that process.	Object Oriented Programming is designed which **focuses on data.**
Structured programming follows **top-down approach**.	Object oriented programming follows **bottom-up approach**.
Structured Programming is also known as **Modular Programming** and a subset of **procedural programming language**.	Object Oriented Programming supports **inheritance, encapsulation, abstraction, polymorphism**, etc.
In Structured Programming, Programs are divided into small self-contained **functions**.	In Object Oriented Programming, Programs are divided into small entities called **objects**.
Structured Programming is **less** secure as there is	Object Oriented Programming is more secure as

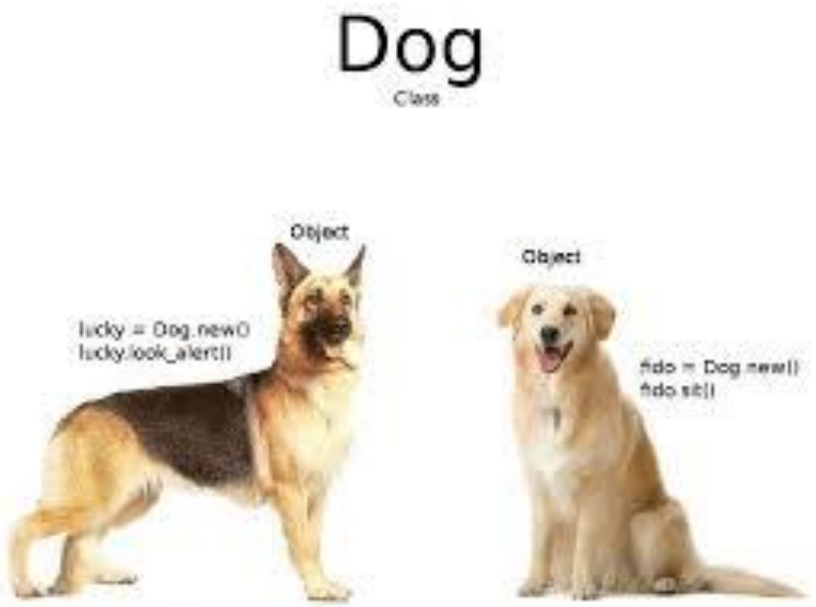
no way of **data hiding**.	having data hiding feature.
Structured Programming can solve **moderately** complex programs.	Object Oriented Programming can solve any **complex** programs.
Structured Programming provides **less reusability**, more function dependency.	Object Oriented Programming provides more reusability, less function **dependency**.
Less abstraction and less flexibility.	More abstraction and more **flexibility**.

Classes and Objects

- **Classes**
  - Like **blueprint** of objects
  - Contain **methods** and **data**
- **Objects**
  - Are **instances (result, output)** of class
  - Create using the **new** keyword
  - Have actions







Class

Object

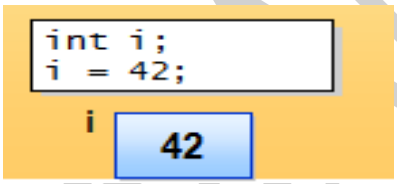
Example

- The **Duck *class*** defines what a **duck is and what it can do.**
    - **Duck** class that **has certain behaviors**, such as **walking**, **quacking**, **flying**, and **swimming**—and **specific properties**, such as **height**, **weight**, and **color**
- A **Duck** *object* is a **specific duck** that has a specific weight, color, height, and behavioral characteristics.
- The duck that you feed is a duck object.

Value Types & Reference Type

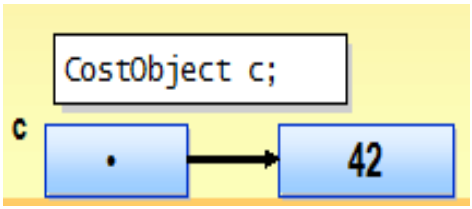
Value Types

- Directly **contain data**
- Stored on the **stack**
- Must be **initialized**
- **Cannot be null**
- An int is a value type
- Example:



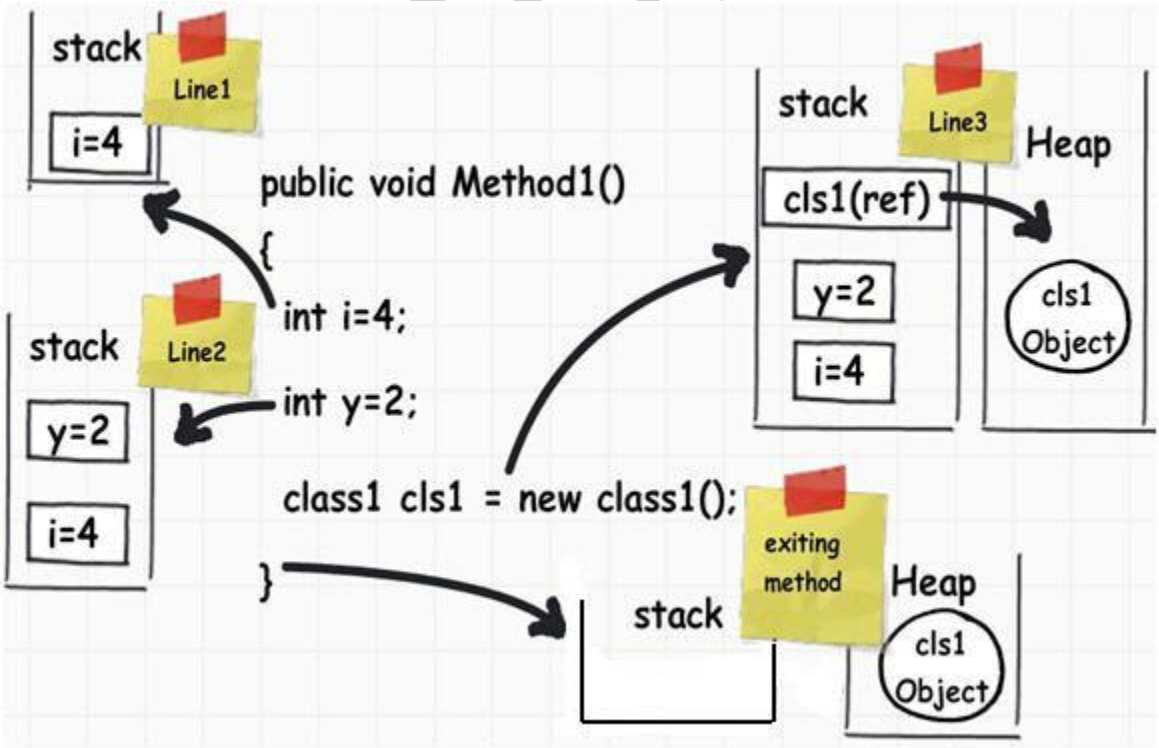
Reference Types

- Contain a **reference to the data**
- Stored on the **heap**
- Declared using **new** key word
- .NET **garbage collection** handles destruction
- A class is a reference type
- Example



Example:

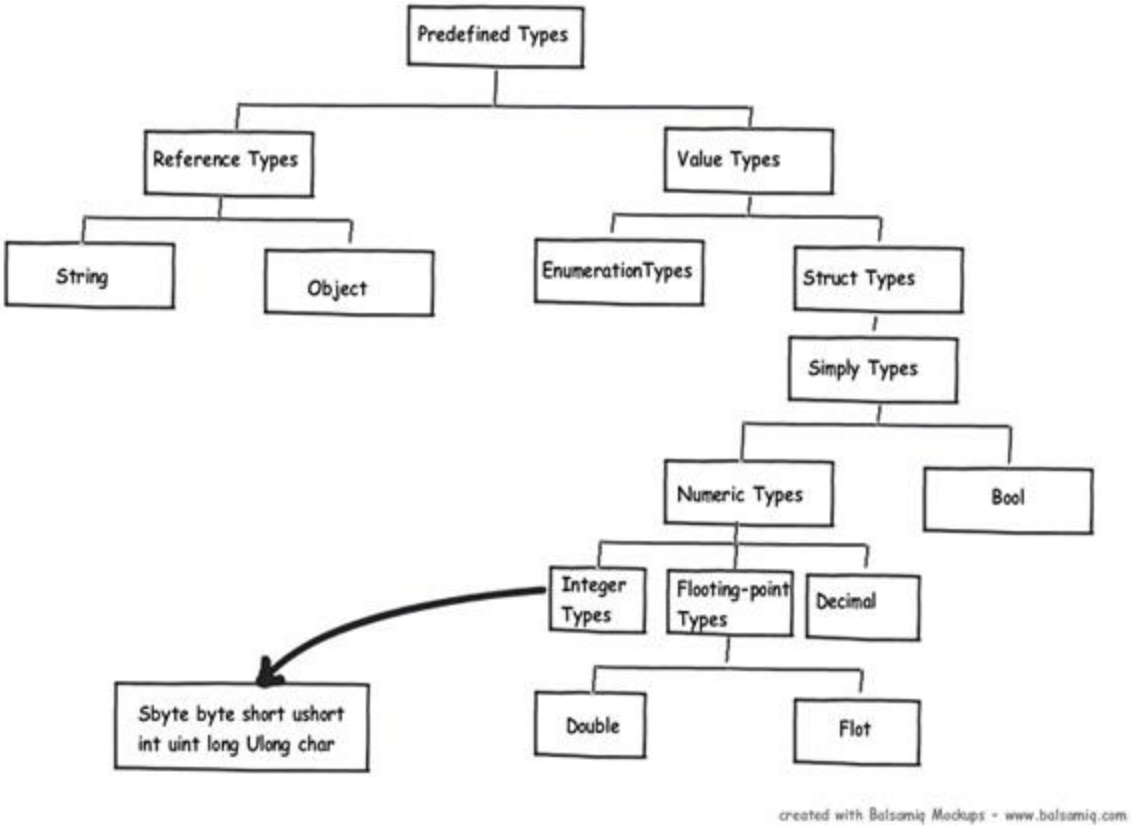
```
public void Method1()
{
    // Line 1
    int i = 4;

    // Line 2
    int y = 2;

    //Line 3
    class1 cls1 = new class1();

}
```



```
struct Point
{
    private int x, y;             // private fields

    public Point(int x, int y)   // constructor
    {
        this.x = x;
        this.y = y;
    }

    public int X                  // property
    {
        get { return x; }
        set { x = value; }
    }

    public int Y
    {
        get { return y; }
        set { y = value; }
    }
}


public class Coordinate
{
    public int X { get; set; }
    public int Y{ get; set; }
}


class Program
{

    static void Main(string[] args)
    {
        //Example of value type
        Point point1 = new Point();
        point1.X = 5;
        point1.Y = 6;

        Console.WriteLine("Point X = {0} Y={1}",point1.X, point1.Y);

        Point point2 = new Point();
        point2 = point1;

        Console.WriteLine("Point X = {0} Y={1}", point1.X, point1.Y);
        Console.WriteLine("Point X = {0} Y={1}", point2.X, point2.Y);

        point2.X = 7;
        point2.Y = 8;

        Console.WriteLine("Point X = {0} Y={1}", point1.X, point1.Y);
        Console.WriteLine("Point X = {0} Y={1}", point2.X, point2.Y);
```

```
            //Example of reference type
            Coordinate cor1 = new Coordinate();
            cor1.X = 9;
            cor1.Y = 10;

            Console.WriteLine("Point X = {0} Y={1}", cor1.X, cor1.Y);

            Coordinate cor2 = new Coordinate();
            cor2 = cor1;

            Console.WriteLine("Point X = {0} Y={1}", cor1.X, cor1.Y);
            Console.WriteLine("Point X = {0} Y={1}", cor2.X, cor2.Y);
            cor2.X = 11;
            cor2.Y = 12;

            Console.WriteLine("Point X = {0} Y={1}", cor1.X, cor1.Y);
            Console.WriteLine("Point X = {0} Y={1}", cor2.X, cor2.Y);

            Console.ReadKey();
        }
    }
```

C# data type hierarchy



Difference between structure and class

Structs differ from classes in several important ways:

- **Structs** are **value types.**
- All **struct** types implicitly inherit from the class **System.ValueType**.
- Assignment to a variable of a **struct** type creates a **copy** of the value being assigned.
- 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**.
- Boxing and unboxing operations are used to convert between a struct type and object.
- The meaning of this is different for structs.
- Instance field declarations for a struct are not permitted to include variable initializers.
- A **struct** is not permitted to declare a **parameterless instance constructor**.
- A **struct** is not permitted to declare a **destructor**.

Choosing Between Class and Struct

✓ **CONSIDER** defining a struct instead of a class if instances of the type are **small** and **commonly short-lived** or are commonly embedded in other objects.

X AVOID defining a struct unless the type has all of the following characteristics:

- It logically represents a single value, similar to primitive types (**int**, **double**, etc.).
- It has an instance size under 16 bytes.
- It is immutable (changeable).
- It will not have to be boxed frequently.

In all other cases, you should define your types as classes

Boxing and Unboxing

- **Boxing**
  - Treat value types like reference types
  - Example: `object` boxedValue = (`object`) x;
  - When we move a value type to reference type, data is moved from the stack to the heap.
- **Unboxing**
  - Treat reference types like value types
  - Example: int y = (int) boxedValue;
  - When we move a reference type to a value type, the data is moved from the heap to the stack.

```
class Program
{
    static void Main(string[] args)
    {
        List<object> mixedList = new List<object>();

        mixedList.Add("First Group:");

        // Add some integers to the list.
        for (int j = 1; j < 5; j++)
        {
            mixedList.Add(j);
        }

        // Add another string and more integers.
        mixedList.Add("Second Group:");
        for (int j = 5; j < 10; j++)
        {
            mixedList.Add(j);
        }

        // Display the elements in the list. Declare the loop variable by
        // using var, so that the compiler assigns its type.
        foreach (var item in mixedList)
```

```
            {
                Console.WriteLine(item);
            }

            var sum = 0;
            for (var j = 1; j < 5; j++)
            {

                sum += (int)mixedList[j] * (int)mixedList[j];
            }

            // The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
            Console.WriteLine("Sum: " + sum);

            int value = 5;
            Console.WriteLine(value);

            object boxedValue = (object)value;

            Console.WriteLine(boxedValue);

            int unboxedValue = (int)boxedValue;

            Console.WriteLine(unboxedValue);

            Console.ReadKey();

        }
    }
```

Chapter 9: Class and Object

Define class and create object

- **How to define class**

```
public class Investor
 {
     public string firstName;
     public string lastName;
     public double purchasePower;
}
```

- How to create an object
    - Example: `Investor objInvestor = new Investor();`
- How to access class variable
    - Example: objInvestor.firstName = “Mahedee”;

Example:

```
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int YearOfExp { get; set; }

    public double GetSalary()
    {
        if (this.YearOfExp > 5)
            return 70000;
        else
            return 40000;
    }


}

class Program
{
    static void Main(string[] args)
    {
        Employee objEmployee = new Employee ();
        objEmployee.FirstName = "Asifur";
        objEmployee.LastName = "Rahman";
        objEmployee.YearOfExp = 7;

        Console.WriteLine("Salary: " + objEmployee.GetSalary());
        Console.ReadKey();
    }
}
```

Object destruction

- When you **create an object**, you are actually **allocating some space in memory** for that object.
- .NET Framework provides an **automatic memory management feature** called ***garbage collection***

Garbage collection

- The garbage collector monitors **the lifetime of objects** and **frees the memory** that is allocated to them when they are **no longer being referenced**.
- By working in the .NET Framework, a programmer **no longer needs to worry** about **de-allocation** and **destruction** of objects in memory.
- When the garbage collector locates objects that are **no longer being referenced,** it implicitly executes the **termination code** that de-allocates the memory and returns it to the pool.
- The garbage collector **does not operate on a predictable schedule**.
- It can run at **unpredictable intervals, usually whenever memory becomes low.**
- Occasionally, you may want to dispose of your objects in a deterministic manner.

- For example, if you must release **scarce resources** over which there may be contention, such as terminating a **database connection** or a communication **port,** you can do so by using the **IDisposable** interface.

Organize classes using namespace

- Consider your video or Movie folder hierarchy
- A ***namespace* is an organizational system** that is used to **identify groups** of **related** classes
- To create a namespace, you simply type the **keyword namespace** followed by a name.
- It is recommended that you use **Pascal case for namespaces**.
- You use namespaces **to organize** classes into a **logically related hierarchy.**
- As an **external way to avoid name clashes (collisions)** between your code and other applications

- **Declaring namespace**

```
namespace Bluechip
{
    public class Investor
    {
        public string firstName;
        public string lastName;
        public double purchasePower;
    }
}
```

- Nested namespaces

```
namespace Bluechip
{
    namespace InvestorSetup
    {
        public class Investor
        {
            //to do
        }
    }
}
```

- **The using namespace**

```
using Bluechip
using Bluechip.InvestorSetup
```

Example: Demo - Namespace and nested namespace

Accessibility and Scope

- Access modifiers are **used to define the accessibility level** of class **members**
    - `public`: Access **not** limited
    - `private`**:** Access limited to the **containing class**
    - `internal`: Access is limited to **this assembly**: classes within the same assembly can access the member.
    - `protected`**:** Access limited to **the containing class** and to types **derived** from the containing class
    - `protected internal:` Access is limited to the **containing class**, **derived classes**, or to classes **within the same assembly** as the containing class.

An ***assembly*** is the **collection of files** that make up a program.

Some rules

The following rules apply:

- Namespaces are always (implicitly) public.
- Classes are always (implicitly) internal.
- Class members are private by default.
- Only one modifier can be declared on a class member. Although ***protected internal*** is two words, it is **one access modifier**.
- The **scope of a member** is never larger than that of its containing type.

Demo of Access Modifier

```
using System;

namespace TestNamespace.Test
{
        public class Lion
        {
            public int age;
            private int weight;
        }

    class ClassMain
    {
        static void Main(string[] args)
        {
            Lion zooLion = new Lion();
            zooLion.age = 7;
```

```
            // the following line causes a compilation error
            zooLion.weight = 200;
        }
    }
}
```

Compiling the preceding code produces the following error:

```
'TestNamespace.Test.ClassMain.Lion.weight' is inaccessible due to its protection level
```

Example internal Access modifier

Assembly1

```
namespace ClassLibrary1
{
    public class CompanyInfo
    {
        public string CompanyName {get; set;}
        internal int NumberOfEmployee { get; set; }
    }

    public class CompanySelector
    {
        //CompanyInfo objCompany = new CompanyInfo();
        public CompanyInfo GetCompanyInfo()
        {
            CompanyInfo objCompanyInfo = new CompanyInfo();
            objCompanyInfo.CompanyName = "Leadsoft Bangladesh Limited";
            objCompanyInfo.NumberOfEmployee = 500;
            return objCompanyInfo;
        }
    }
}
```

Console Application (Assembly 2)

```
namespace ConsoleApplication14
{
    class Program
    {
        static void Main(string[] args)
        {
            CompanySelector objCompanySelector = new CompanySelector();
            CompanyInfo objCompanyInfo = objCompanySelector.GetCompanyInfo();

            Console.WriteLine(objCompanyInfo.CompanyName);
            //Console.WriteLine(objCompanyInfo.NumberOfEmployee);
            Console.ReadKey();
        }
    }
}
```

Example : Protected Internal

Class Library 1:

```
namespace ClassLibrary1
{
    public class CompanyInfo
    {
        public string CompanyName {get; set;}
        internal int NumberOfEmployee { get; set; }
        protected int NumberOfAchitect { get; set; }
        protected internal int NumberOfSofwareEngineers { get; set; }

    }

    public class CompanySelector
    {
        public CompanyInfo GetCompanyInfo()
        {
            CompanyInfo objCompanyInfo = new CompanyInfo();
            objCompanyInfo.CompanyName = "Leadsoft Bangladesh Limited";
            objCompanyInfo.NumberOfEmployee = 500;
            objCompanyInfo.NumberOfSofwareEngineers = 250;

            return objCompanyInfo;
        }
    }

}
```

Console Application:

```
namespace ConsoleApplication14
{
    public class MyCompany : CompanyInfo
    {
        public MyCompany()
        {
            base.CompanyName = "Leads Corporation Ltd";
            base.NumberOfAchitect = 20;
            base.NumberOfSofwareEngineers = 250;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            MyCompany objMyCompany = new MyCompany();

            Console.WriteLine(objMyCompany.CompanyName);


            CompanySelector objCompanySelector = new CompanySelector();
            CompanyInfo objCompanyInfo = objCompanySelector.GetCompanyInfo();
```

```
            Console.WriteLine(objCompanyInfo.CompanyName);
            //Console.WriteLine(objCompanyInfo.NumberOfArchitect);
            //Console.WriteLine(objCompanyInfo.NumberOfEmployee);
            Console.ReadKey();

        }
    }
}
```

Declaring Method

- A method is a class **member** that is used to **define the actions**
- It can be **performed by that object** or **class**.
- Syntax of declaring method:

```
<Access Modifier> <Return Type> <Method Name> (Parameter List)
{
    //Method Body
}
```

  - **Access modifiers / Specifier**: This determines the visibility of a variable or a method from another class.
  - **Return type**: A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**.
  - **Method name**: Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
  - **Parameter list**: Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
  - **Method body**: This contains the set of instructions needed to complete the required activity.

- The following rules apply to methods:
  - In the method declaration, you must always specify a **return type**. If the method is not designed to return a value to the caller, you specify a return type of **void**.
  - Even if the method takes **no arguments, you must include a set of empty parentheses** after the method name.
  - When calling a method, you must match the **input parameters of the method exactly**, including the return type, the number of parameters, their order, and their type. **The method name and parameter list is known as the method *signature***

- The following are recommendations for naming methods:
  - The name of a method should represent the **action** that you want to carry out. For this reason, methods usually have **action-oriented names**, such as **WriteLine** and **ChangeAddress**.
  - Methods should be named using **Pascal case.**

- When you call a method, the execution jumps to that method and it executes until either a **return** statement or the end of the method is reached
- **Declare Mathod:**

```
class Lion
{
    private int weight;

    public bool IsNormalWeight()
    {
        if ((weight < 100) || (weight > 250))
        {
            return false;
        }
        return true;
    }
    public void Eat() { }

    public void SetWeight(int weight)
    {
        this.weight = weight;
    }

    public int GetWeight()
    {
        return weight;
    }
}
```

- **Invoke method:**

```
        Lion bigLion = new Lion();
        bigLion.SetWeight(50);

        if (bigLion.IsNormalWeight() == false)
        {
            Console.WriteLine("Lion weight is abnormal");
        }
```

Example:

```
class Lion
{
    private int weight = 50;

    public bool IsNormalWeight()
    {
        if ((weight < 100) || (weight > 250))
        {
            return false;
        }
        return true;
```

```
        }
        public void Eat() { }


        public int GetWeight()
        {
            return weight;
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Lion bigLion = new Lion();
            if (bigLion.IsNormalWeight() == false)
            {
                Console.WriteLine("Lion weight is abnormal");
            }

            Console.ReadKey();
        }
    }
```

this keyword

The **this** keyword is used to refer to the **current instance** of an object. When **this** is used within a method, it allows you to refer to the members of the object.

For example, the **GetWeight** method can be modified to use **this** as follows:

```
public int GetWeight() {
   return this.weight;
}
```

Pass Parameter to Method

Passing by value

Example:

```
        class Lion
        {
            private int weight;

            public bool IsNormalWeight()
            {
```

```
            if ((weight < 100) || (weight > 250))
            {
                return false;
            }
            return true;
        }
        public void Eat() { }

        public void SetWeight(int weight)
        {
            this.weight = weight;
        }

        public int GetWeight()
        {
            return weight;
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Lion bigLion = new Lion();
            bigLion.SetWeight(50);

            if (bigLion.IsNormalWeight() == false)
            {
                Console.WriteLine("Lion weight is abnormal");
            }

            Console.ReadKey();
        }
    }
```

Passing by reference

- **Methods return only a single value**, but sometimes you want a method to modify or **return multiple values.**
    - You can achieve this by **passing the method a *reference*** to the variable that you want to modify

- You declare that a parameter is a **reference parameter** by using the **ref** keyword.
    - Use the **ref** keyword in the parameter list to indicate to the **compiler that the value is being passed by reference**
    - You also must use the **ref** keyword when you **call the method**.

- **Definite** assignment
- Using **out parameter** keyword
    - **Allow you to initialize variable** in method
- Use if you want a method to modify or r**eturn multiple values**

- Achieve this by passing the method a ***reference***

Example 1:

```
class Zoo
{
    private int streetNumber = 123;
    private string streetName = "High Street";
    private string cityName = "Dhaka";
    public void GetAddress(ref int number, ref string street, ref string city)
    {
        number = streetNumber;
        street = streetName;
        city = cityName;
    }
}

class ClassMain
{
    static void Main(string[] args)
    {
        Zoo localZoo = new Zoo();

        int zooStreetNumber = 0 ;
        string zooStreetName = null;
        string zooCity = null;

        localZoo.GetAddress(ref zooStreetNumber, ref zooStreetName, ref zooCity);
        Console.WriteLine(zooCity);
        // Writes "Dhaka" to a console
        Console.ReadKey();
    }
}
```

- Even if you have a variable that you know **will be initialized within a method**, **definite assignment requires** that **you initialize the variable before it can be passed to a method**.
- For example, in the preceding code, the lines that initialize **zooStreetNumber**, **zooStreetName**, and **zooCity** are required to make the code compile, even though the intent is to initialize them in the method.

- By using the **out** keyword, **you can eliminate the redundant initialization**.
- Use the **out** keyword in situations where you want to inform the compiler that variable **initialization is occurring within a method**.
- When you use the **out** keyword with a variable that is being passed to a method, you can pass an uninitialized variable to that method.

Example 2:

```
class Zoo
{
    private int streetNumber = 123;
    private string streetName = "High Street";
    private string cityName = "Dhaka";

    public void GetAddress(out int number,
                           out string street,
                           out string city)
    {
        number = streetNumber;
        street = streetName;
        city = cityName;
    }
}

class ClassMain
{
    static void Main(string[] args)
    {
        Zoo localZoo = new Zoo();
        // note these variables are not initialized
        int zooStreetNumber;
        string zooStreetName;
        string zooCity;
        localZoo.GetAddress(out zooStreetNumber,
                           out zooStreetName,
                           out zooCity);

        Console.WriteLine(zooCity);
        // Writes "Dhaka" to a console

        Console.ReadKey();
    }
}
```

How to Pass a Reference Type

- When you pass a **reference** type to a method, the method can **alter the actual object**.

Example:

```
class Address
{
    public int number;
    public string street;
    public string city;
}

class Zoo
{
    private int streetNumber = 123;
    private string streetName = "High Street";
```

```
    private string cityName = "Dhaka";

    public void GetAddress(Address zooAddress)
    {
        zooAddress.number = streetNumber;
        zooAddress.street = streetName;
        zooAddress.city = cityName;
    }
}

class ClassMain
{
    static void Main(string[] args)
    {
        Zoo localZoo = new Zoo();
        Address zooLocation = new Address();
        zooLocation.city = "Chittagong";

        localZoo.GetAddress(zooLocation);

        Console.WriteLine(zooLocation.city);
        // Writes "Dhaka" to a console

        Console.ReadKey();
    }
}
```

Method Overloading

- Method overloading is **a language feature** that enables you to create **multiple methods** in **one class** that have the **same name** but that take **different signatures**
- By overloading a method, you **provide the users of your class with a consistent name** for an action while also providing them with **several ways to apply that action**.
- Overloaded methods are a good way for you to **add new functionality to existing code.**
- **Why overload?**
    - Use overloading when you have **similar methods that require different parameters.**
    - Overloaded methods are a **good way for you to add new functionality to existing code.**
    - Use overloaded methods only to implement methods that provide **similar functionality.**

Example 1:

```
class Zoo
{
    public void AddLion(Lion newLion)
    {
        // Place lion in an appropriate exhibit
    }
    public void AddLion(Lion newLion, int exhibitNumber)
```

```
        {
            // Place the lion in exhibitNumber exhibit
        }
    }

    Zoo myZoo = new Zoo();
    Lion babyLion = new Lion();
    myZoo.AddLion(babyLion);
    myZoo.AddLion(babyLion, 2);
```

Example 2:

```
    class Program
    {
        static void Main(string[] args)
        {
            ZooCustomer objZooCustomer = new ZooCustomer();
            objZooCustomer.SetInfo("mahedee.net", "mahedee.hasan@gmail.com");
            objZooCustomer.SetInfo("bdzoo.com", "bdzoo@gmail.com","Tiger");
        }
    }

    class ZooCustomer
    {
        private string name;
        private string email;
        private string favoriteAnimal;

        public void SetInfo(string webName,
                            string webEmail,
                            string animal)
        {
            name = webName;
            email = webEmail;
            favoriteAnimal = animal;
        }

        public void SetInfo(string webName, string webEmail)
        {
            name = webName;
            email = webEmail;
        }
    }
```


Constructor

- Constructors are **special methods** that implement the actions that **are required to initialize an object.**
    - Have the **same name** as the name of the **class**
    - **Default constructor** takes no parameter

Example:

```
public class Lion
{
    private string name;
    public Lion()
    {
        Console.WriteLine("Constructing Lion");
    }
    public Lion(string newLionName)
    {
        this.name = newLionName;
    }
}


class Program
{
    static void Main(string[] args)
    {
        Lion babyLion = new Lion();
        Console.WriteLine("Made a new Lion object");
        Console.ReadKey();
    }
}
```

Output:
Constructing Lion
Made a new Lion object

Constructor Parameters

- Constructors **can take parameters**, like any other method
- **Different way to initialize an object**
- When **an object is created**, the instance members in the class are **implicitly initialized.**
    - For example, in the following code, **zooName** is initialized to **null**:

```
class Zoo
{
    public string zooName;
 }
```

Example:

```
public class Lion
{
    private string name;

    public Lion()
```

```
        {
            this.name = "Lion name not found!";
        }

        public Lion(string newLionName)
        {
            this.name = newLionName;
        }

        public void LionName()
        {
            Console.WriteLine(name);
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Lion babyLion = new Lion("Leo is a baby lion");
            babyLion.LionName();

            Lion noNameLion = new Lion();
            noNameLion.LionName();

            Console.ReadKey();
        }
    }
```

readonly variable

- When you use the **readonly modifier** on a member **variable**, y**ou can only assign it a value when the class or object initializes**
    - Either by **directly assigning the member** variable a value, or **by assigning it in the constructor.**
- Use the **readonly** modifier **when a const keyword is not appropriate** because **you are not using a literal value—meaning** that the actual value of the variable is not known at the time of compilation.

Example:

```
    class Zoo
    {
        private int numberAnimals;
        public readonly decimal admissionPrice;

        public Zoo(int noOfAnimal)
        {
            this.numberAnimals = noOfAnimal;
            // Get the numberAnimals from some source...
```

```
            if (numberAnimals > 50)
            {
                admissionPrice = 25;
            }
            else
            {
                admissionPrice = 20;
            }
        }

        public void AdmissionPrice()
        {
            Console.WriteLine("Admission cost is: " + admissionPrice);
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Zoo zoo = new Zoo(60);
            zoo.AdmissionPrice();

            Console.ReadKey();

        }
}
```

Constructor Overloading

- Create **multiple constructor** that have **same name but different signatures**
- It is often useful to overload a constructor to allow **instances to be created in more than one way.**

Example 1:

```
class Lion
{
    private string name;
    private int age;

    public Lion(string theName, int theAge)
    {
        name = theName;
        age = theAge;
    }

    public Lion(string theName)
    {
        name = theName;
    }

    public Lion(int theAge)
```

```
        {
            age = theAge;
        }

        public void DisplayLionInfo()
        {
            Console.WriteLine("Lion name: "+name+"\nLion age: " + age);
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Lion adoptedLion = new Lion("Leo", 3);
            adoptedLion.DisplayLionInfo();

            Lion otherAdoptedLion = new Lion("Fang");
            otherAdoptedLion.DisplayLionInfo();

            Lion newbornLion = new Lion(1);
            newbornLion.DisplayLionInfo();

            Console.ReadKey();
        }
    }
```

Example 2:

```
    public class Lion
    {
        private string name;
        private int age;

        public Lion() : this("unknown", 0)
        {
            Console.WriteLine("Default {0}", name);
        }

        public Lion(string theName, int theAge)
        {
            name = theName;
            age = theAge;
            Console.WriteLine("Specified: {0}", name);
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Lion adoptedLion = new Lion();
            Console.ReadKey();
        }
```

```
}
```

Static Class Members

- Static members **belong to the class**, rather than an instance.
- Static constructors are **used to initialize a class**.
- **Initialize before an instance of the class** is created.
- Shared by all instance of the class
- Classes can have static members, such as properties, methods and variables.
- Because static members belong to the class, rather than an instance, they are accessed through the **class**, not through an **instance of the class.**

Example:

```
public class Company
{
    public static string companyName = "Leadsoft Bangladesh Limited.";
}


class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(Company.companyName);
        Console.ReadLine();
    }
}
```

Example 2:

```
public class InterestCalculator
{
    public static double CalculateInterest(int n, double p, double r)
    {
        return n * p * r;
    }
}

class Program
{
    static void Main(string[] args)
```

```
        {
            double interest = InterestCalculator.CalculateInterest(5, 50000.0, 0.10);
            Console.WriteLine("Simple Interest: " + interest);
            Console.ReadLine();
        }
    }
```

Static Constructor

- **Instance** constructors are used to **initialize an object**
- Static constructors are used to **initialize a class**
- Will only **ever** be **executed once**
- Run **before the first object** of that type is created.
- Have **no parameter**
- Do not take an access modifier
- May **co-exist** with a class constructor
- **Member variable must be static** for the static constructor to be able to assign a value to it.

Syntax:

```
class Lion
{
    static Lion()
    {
        // class-specific initialization
    }
}
```

Example:

```
class RandomNumberGenerator
{
    private static Random randomNumber;
    public static string AuthorName { get; set; }

    public RandomNumberGenerator(String msg)
    {
        Console.WriteLine(msg);
        //Constructor for object
    }
```

```
        //Static constructor
        static RandomNumberGenerator()
        {
            AuthorName = "Mahedee Hasan";
            randomNumber = new Random();
        }

        public int Next()
        {
            return randomNumber.Next();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            RandomNumberGenerator randomNumber
                = new RandomNumberGenerator("Generate 10 Random Number");

            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine(randomNumber.Next());
            }

            Console.WriteLine("Author Name: " + RandomNumberGenerator.AuthorName);
            Console.ReadKey();
        }
    }
```

Structure Vs Object Oriented Design in a nutshell

Structure Design	Object Oriented Design
Process centered	**Object** centered
Reveals data	Hide data
Single unit	Modular unit
One time use	Reusable
Ordered algorithm	No ordered algorithm

Chapter 10: Major features of Object Oriented Programming

Object oriented programming has three major features of principles

- **Encapsulation**. **Capability to hide data** and instructions inside an object.
- **Inheritance**. Ability to create one object from another.
- **Polymorphism**. The design of new classes is based on a single class.

Encapsulation

- **Grouping related piece of information** and processes into **self-contained unit.**
    - Makes it **easy to change**
        - The way **things work under the cover** without changing the way users interact.
- **Hiding** internal details.
    - Makes your object easy to use.

What Are Properties?

- **Properties are methods** that **protect access to class members**.
- Properties are **class members** that **provide access to elements** of an object or class.
- **Protect access** to the **state** of object.
- It likes **fields**, but they operate much like methods.

- The **get** and **set** statements are called **accessors**.
- **Fields can't be used in Interfaces** so properties are the solution.

Syntax:

```
private double balance;
public double Balance
{
    get
    {
        return balance;
    }
    set
    {
        balance = value;
    }
}
```

Example:

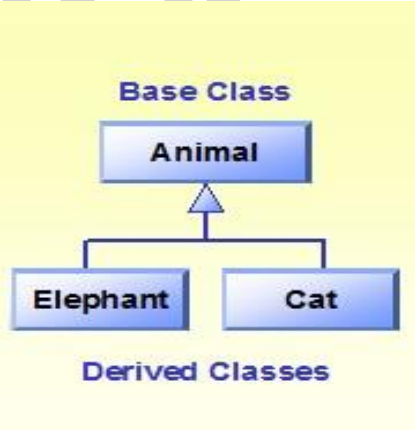
```
public class Employee
{
    private double salary;
    private double taxRate = 0.05;


    public string Name { get; set; }
    public double YearOfExp { get; set; }

    public double YearlyMedicalAllowance { get; private set; }

    public Employee()
    {
        this.YearlyMedicalAllowance = 30000;
    }

    public double Salary
    {
        get { return salary; }
        set
        {
            if (value > 200000)
                salary = value - value * taxRate;
            else
                salary = value;
        }
    }

}

class Program
{
    static void Main(string[] args)
```

```
        {
            Employee objEmployee = new Employee();
            objEmployee.Name = "Rafiqul Islam";
            objEmployee.YearOfExp = 7;
            objEmployee.Salary = 300000;

            Console.WriteLine(objEmployee.Name);
            Console.WriteLine("Salary: " + objEmployee.Salary);
            Console.WriteLine("Yearly Madical Allowance: " +
objEmployee.YearlyMedicalAllowance);

            Console.ReadLine();
        }
    }
```

Inheritance

- Inheritance specifies an **is-a kind** of relationship
- Derived classes **inherits properties and methods from base class**
    - Allowing **code reuse**
- Derived classes become more **specialized**.
- The object-oriented principle of inheritance enables you to create a generalized class and then derive more specialized classes from it.
    - The **general class** is referred to as the ***base* class**.
    - A more **specific class** is referred to as the ***derived* class**.

- The primary **benefit** of inheritance is **code reuse**



Problem:

```
    public class Antelope
    {
        public bool IsSleeping;
        public void Sleep() { }
        public void Eat() { }
    }
```

```
public class Lion
{
    public bool IsSleeping;
    public void Sleep() { }
    public void EatAntelope() { }
    public void StalkPrey() { }
}

public class Elephant
{
    public bool IsSleeping;
    public void Sleep() { }
    public int CarryCapacity;
    public void Eat() { }
}



class Program
{
    static void Main(string[] args)
    {
        Elephant e = new Elephant();
        e.Sleep();
    }
}
```

- ***Duplication of code***. For example, the **Sleep method** must be implemented multiple times.
- ***User confusion*.** The **eating methods have different names** in different objects.

Example:

```
public class Animal
{
    public bool IsSleeping;
    public void Sleep()
    {
        Console.WriteLine("Sleeping");
    }
    public void Eat() { }
}

public class Antelope : Animal
{
}

public class Lion : Animal
{
    public void StalkPrey() { }
}

public class Elephant : Animal
{
```

```
    public int CarryCapacity;
}

class Program
{
    static void Main(string[] args)
    {
        Elephant e = new Elephant();
        e.Sleep();
        Console.ReadKey();
    }
}
```

Call Base constructor from derived class

- When you create an object, the **instance constructor** is executed.
- When you create an **object from a derived class**, the **instance constructor for the base class is executed first**, and then the instance constructor for the derived class is executed.

Order of execution

Example:

```
public class Animal
{
    public Animal()
    {
        Console.WriteLine("Constructing Animal");
    }
}

public class Elephant : Animal
{
    public Elephant()
    {
        Console.WriteLine("Constructing Elephant");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Elephant e = new Elephant();
        Console.ReadKey();
    }
}
```

Calling specific constructor

- You use the **base** keyword in the **derived class constructor** to call the base class constructor with a **matching signature.**

```
public enum GenderType
{
```

```
        Male,
        Female
    }

    public class Animal
    {
        public Animal()
        {
            Console.WriteLine("Constructing Animal");
        }

        public Animal(GenderType gender)
        {
            if (gender == GenderType.Female)
            {
                Console.WriteLine("Female ");
            }
            else
            {
                Console.WriteLine("Male ");
            }
        }
    }

    public class Elephant : Animal
    {
        public Elephant(GenderType gender)
            : base(gender)
        {
            Console.WriteLine("Elephant");
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Elephant e = new Elephant(GenderType.Female);
            Console.ReadKey();
        }
    }
```

Sealed Class

- You **cannot derive** from a **sealed** class
- Prevents the class from being **overridden** or **extended** by third parties

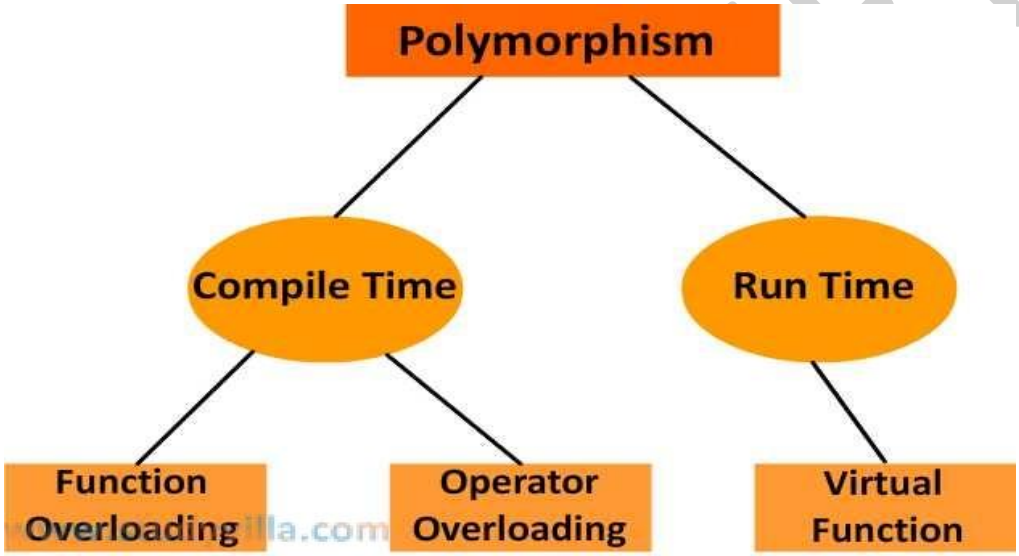
Syntax:

```
    public sealed class Account
    {
       //...
    }
```

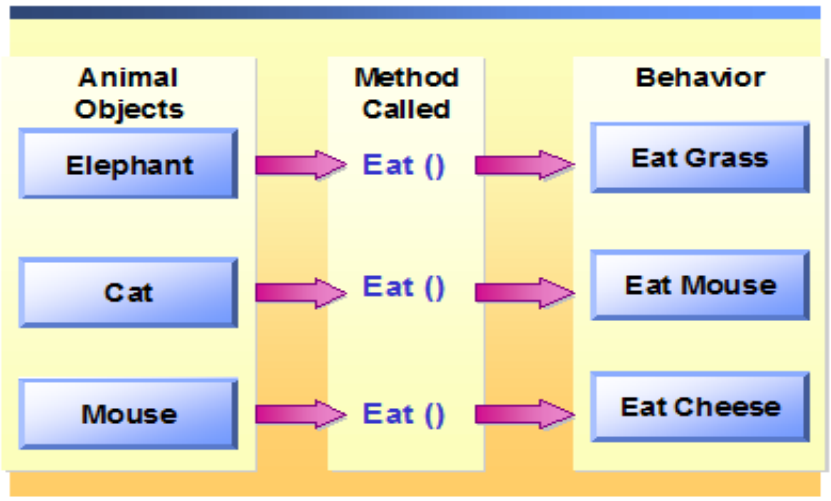
Polymorphism

- Polymorphism is an **object-oriented concept** that enables you
  - To **treat** your **derived classes in a similar manner**, even though they are different.

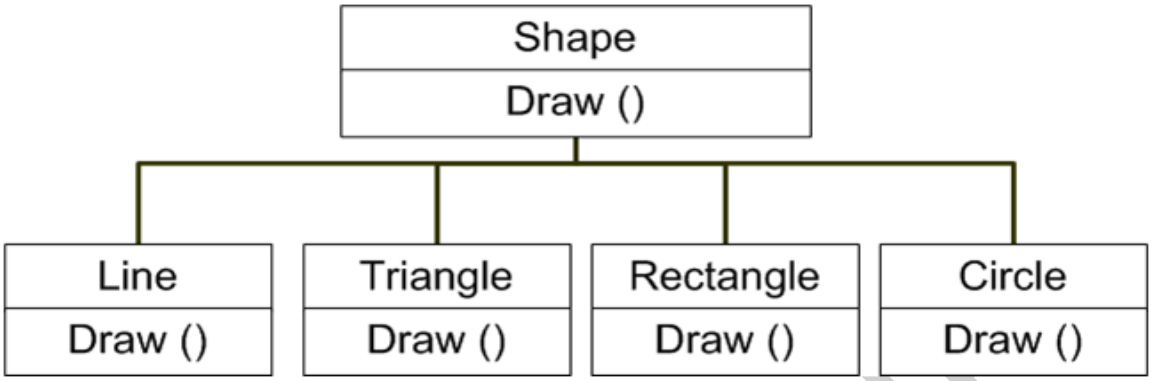
- When you create derived classes, you provide more **specialized functionality**
  - Polymorphism enables you to **treat these new objects in a general way**.



Example 1:



Example 2:



Virtual Methods

- A **virtual method** is one whose **implementation can be replaced** by a method in a derived class.
- Use the keyword **virtual, in the base class method**
- **Use the override keyword, in the derived class method.**
- When you override a virtual method, the overriding method **must have the same signature** as the virtual method.

Example:

```
public class Animal
{  // base class
    public Animal() { }
    public void Sleep() { }
    public bool IsHungry = true;
    public virtual void Eat()
    {
        Console.WriteLine("Eat something");
    }
}

public class Elephant : Animal
{
    public int CarryCapacity;
    public override void Eat()
    {
        Console.WriteLine("Eat grass");
    }
}

public class Mouse : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Eat cheese");
```

```
    }
}

public class Cat : Animal
{
    public void StalkPrey() { }
    public override void Eat()
    {
        Console.WriteLine("Eat mouse");
    }
}

public class WildLife
{
    public WildLife()
    {
        Elephant objElephant = new Elephant();
        Mouse objMouse = new Mouse();
        Cat objCat = new Cat();

        FeedingTime(objElephant);
        FeedingTime(objMouse);
        FeedingTime(objCat);
    }

    public void FeedingTime(Animal objAnimal)
    {
        //Notice use of polymorphism here
        if (objAnimal.IsHungry)
        {
            objAnimal.Eat();
        }
    }

    static void Main(string[] args)
    {
        WildLife objWildLife = new WildLife();
        Console.ReadKey();
    }
}
```

Use Base Class Members from a Derived Class

- The **base** keyword is used in derived classes **to access members of the base class.**

```
public class Animal
{
    public virtual void Eat()
    {
        Console.WriteLine("Eat something");
    }
}

public class Cat : Animal
{
```

```
        public void StalkPrey() { }
        public override void Eat()
        {
            base.Eat();
            Console.WriteLine("Eat small animals");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Cat objCat = new Cat();
            objCat.Eat();
           Console.ReadKey();

        }
    }
```

Abstract Methods and Classes

- An **abstract class** is a **generic base** class
  - Contains an **abstract method** that **must be implemented** by a derived class.
- An **abstract method has no implementation** in the base class
- Abstract class **can contain non abstract members**
- Any class that contains abstract members **must be abstract**
- Because the **purpose** of an abstract class is to act as a **base class**
  - It is **not possible to instantiate an abstract class** directly
  - Nor can an abstract class be **sealed**.

- **Syntax**
  [*access-modifiers*] **abstract** *return-type method-name* ([*parameters*]) **;**

```
public abstract class Animal
{
    public abstract void Eat();
}
```

Example:

```
public abstract class Animal
{
    public abstract void Eat();
}


public class Mouse : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Eat cheese");
    }
```

```
}


class Program
{
    static void Main(string[] args)
    {
        Mouse objMouse = new Mouse();
        objMouse.Eat();
        Console.ReadKey();
    }
}
```

Abstract class with virtual method

Example:

```
public abstract class Animal
{
    public virtual void Sleep()
    {
        Console.WriteLine("Sleeping");
    }
    public abstract void Eat();
}


public class Mouse : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Eat cheese");
    }
    public override void Sleep()
    {
        Console.WriteLine("Mouse sleeping");
    }
}

public class Dog : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Eat meat");
    }
}


class Program
{
    static void Main(string[] args)
    {
        Mouse objMouse = new Mouse();
        objMouse.Sleep();
        objMouse.Eat();

        Dog objDog = new Dog();
```

```
            objDog.Sleep();
            objDog.Eat();

            Console.ReadKey();
        }
    }
```

Abstract properties

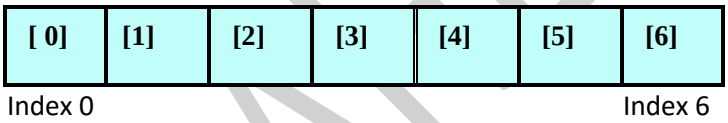
Example:

```
    public abstract class Account
    {
        public abstract double Balance { get; }
        public abstract string AccountHolderName { get; set; }
    }

    public class CurrentAccount : Account
    {
        public override double Balance {get;}
        public override string AccountHolderName {get; set;}
    }
```

Chapter 11: Array and Collection

What is Array?

- **A data structure** that contains a **number of variables** called **element** of the array.
- All the array elements **must be of the same type**.
- Arrays are **zero indexed.**



- Arrays can be:
    - **Single- dimentional**, an array with the **rank one**.
    - **Multi-dimensional**, an array with the **rank more than one**
    - **Jagged**, an array whose **elements are arrays**

Common used array methods

Method	Description
Sort	Sorts the elements in an array

Clear	Sets a range of elements to zero or **null**
Clone	Creates a copy of the array
GetLength	Returns the **length** of a given **dimension**
IndexOf	Returns the **index of the first occurrence** of a value
Length	**Gets the number of elements** in the specified dimension of the array

For more about array method and properties visit: https://msdn.microsoft.com/en-us/library/system.array(v=vs.100).aspx

How to create an array

- **Declare** the array by adding a set of square brackets to the end of the **variable type** of the individual elements

```
int[] MyIntegerArray;
```

- **Instantiate** to create

```
int[] numbers = new int[5];
```

- To create an array of type object

```
object[] animals = new object[100];
```

Initialize and Access Array Members

- **Initialize** an array

```
int[] numbers = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
numbers[4] = 5;
```

- **Accessing** array members

```
string[] animals = { "Elephant", "Cat", "Mouse" };
animals[1] = "cow";
String someAnimal = animals[2];
```

Iterate an Array Using the *foreach* Statement

- Using ***foreach*** statement **repeats** the embedded statement(s) **for each elements** in the arrays.

Example 1:

```
class Program
{
    static void Main(string[] args)
    {
        int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
        foreach (int i in numbers)
        {
            Console.WriteLine(i);
        }

        Console.ReadKey();

    }
}
```

Example 2:

```
public abstract class Animal
{
    abstract public void Eat();
}

public class Lion : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Lion eats meat.");
    }
}

public class Elephant : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Elephant eats vegetation.");
    }
}

class ClassZoo
{
    static void Main(string[] args)
    {
        Lion objLion = new Lion();
        Elephant objElephant = new Elephant();

        Animal[] zoo = new Animal[2];
        zoo[0] = objLion;
        zoo[1] = objElephant;
```

```
            foreach (Animal animal in zoo)
            {
                animal.Eat();
            }
            Console.ReadKey();
        }
    }
```

Use Arrays as Method Parameters

```
    public class MathArray
    {
        public int Sum(int[] numberList)
        {
            int sum = 0;
            foreach (int number in numberList)
            {
                sum += number;
            }
            return sum;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            MathArray objMathArray = new MathArray();
            int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };
            int total = objMathArray.Sum(numbers);
            Console.WriteLine(total);  // 28
            Console.ReadKey();
        }
    }
```

Uses of params keyword

- ***params*** keyword used to pass a **variable number** of **arguments** to method

```
    public class ParamTest
    {
        public int Sum(params int[] listNumbers)
        {
            int total = 0;
            foreach (int number in listNumbers)
            {
                total += number;
            }
            return total;
        }
```

```
    public string Combine(string str1, string str2, params object[] others)
    {
        string combination = str1 + " " + str2;
        foreach (object obj in others)
        {
            combination += " " + obj.ToString();
        }
        return combination;
    }

}

class Program
{
    static void Main(string[] args)
    {
        ParamTest objParamTest = new ParamTest();

        int total = objParamTest.Sum(1, 2, 3, 4, 5, 6, 7);
        Console.WriteLine(total);

        string combo = objParamTest.Combine("One", "two", "three", "four");
        Console.WriteLine(combo);

        combo = objParamTest.Combine("alpha", "beta");
        Console.WriteLine(combo);

        Console.ReadKey();
    }
}
```

Collections

Lists, Queues, Stacks, and Hash Tables

- Lists, Queues, Statcks and Hash Tables are **common way to manage data** in an application
- **List**: A **collection** that allows you **access by index**.
    - **Example: An array is a list, an ArrayList is a list**
- **Queue**: **First in first out** collection of objects.
    - Example: Waiting in line at a ticket office.
- **Stack**: **Last in first out** collection of objects.
    - Example: a pile of plates
- **Hash Tables**: Represents a collection of associated **keys and values organized** around the hash code of the key.
    - Example: Dictionary

ArrayList Class

- ArrayList **does not have fixed size**; it grows as needed
- Use **Add(object)** to add an object to the end of the ArrayList
- Use [] to access elements of the ArrayList.
- Use **TrimToSize()** to reduce the size to fit the number of elements in the ArrayList
    - Trimming an empty ArrayList sets the capacity of the ArrayList to the default capacity.
- Use **clear** to remove all the elements
- Can set the capacity explicitly

Common Methods of ArrayList

Method	Use
Add	Adds an object to the end of the ArrayList.
Remove	Removes the first occurrence of a specific object from the ArrayList.
Clear	Removes all elements from the ArrayList.
Insert	Inserts an element into the ArrayList at the specified index.
TrimToSize	Sets the capacity to the actual number of elements in the ArrayList.
Sort	Sorts the elements in the ArrayList.
Reverse	Reverses the elements in the ArrayList.

Example: ArrayList

```
class Program
{
    static void Main(string[] args)
    {
        ArrayList arrList = new ArrayList();
```

```
            arrList.Add(123);
            arrList.Add("Mahedee Hasan");
            arrList.Add(DateTime.Now);
            arrList.Remove(123);
            arrList.Clear();
            arrList.Insert(0, "Hasanur Rahman");
            arrList.Add("Newaz Sharif");
            arrList.Reverse();

            foreach (var item in arrList)
            {
                Console.WriteLine(item.ToString());
            }

            Console.ReadKey();
        }
```

List<T>Class

- Represents a **strongly typed list** of objects that can be accessed by **index.**
- Provides methods to search, sort, and manipulate lists.

Syntax:

```
List<T> lstTObject = new List<T>();
```

Example of List

```
    public class Employee
    {
        public string Name { get; set; }
        public string Designation { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            List<Employee> lstEmployee = new List<Employee>();
            Employee se = new Employee();
            se.Name = "Abdullah Yousuf";
            se.Designation = "Software Engineer";

            lstEmployee.Add(se);

            Employee sse = new Employee();
            sse.Name = "Anamul Haque";
            sse.Designation = "Senior Software Engineer";

            lstEmployee.Add(sse);

            Employee sa = new Employee();
            sa.Name = "Mahedee Hasan";
            sa.Designation = "Software Architect";
```

```
            lstEmployee.Add(sa);

            Employee ssa = new Employee();
            ssa.Name = "Tanvir Reza Haque";
            ssa.Designation = "Senior Software Architect";

            lstEmployee.Add(ssa);

            Employee mgr = new Employee();
            mgr.Name = "Faizur Rahman";
            mgr.Designation = "Manager";

            lstEmployee.Add(mgr);

            foreach (var employee in lstEmployee)
            {
                Console.WriteLine(employee.Name + " Designation: " +
employee.Designation);
            }

            Employee pm = new Employee();
            pm.Name = "Ahsan Kabir";
            pm.Designation = "Project Manager";
            lstEmployee.Insert(1, pm);

            Console.WriteLine(lstEmployee[1].Name + " Designation: " +
lstEmployee[1].Designation);
            Console.WriteLine(lstEmployee.Count);

            lstEmployee.Reverse();
            lstEmployee.RemoveAll(p => p.Name == "Mahedee Hasan");

            foreach (var employee in lstEmployee)
            {
                Console.WriteLine(employee.Name + " Designation: " +
employee.Designation);
            }
            Console.ReadKey();

        }
    }
```

Queues and Stacks

- **Queues**: **first in, first out**
    - Enqueue **places** object in the Queue
    - Dequeue **removes** objects from the Queue.

- **Stacks**: **last in, first out**
    - **Push** places object in the stack
    - **Pop** removes object from the stack
    - **Counts** get the number of objects contained in a stack or queue

Example: Queue

```
class Message
{
    private string messageText;

    public Message(string str)
    {
        messageText = str;
    }
    public override string ToString()
    {
        return messageText;
    }
}

class Buffer
{
    private Queue messageBuffer;

    public void SendMessage(Message msg)
    {
        messageBuffer.Enqueue(msg);
    }
    public void ReceiveMessage()
    {
        Message message = (Message)messageBuffer.Dequeue();
        Console.WriteLine(message.ToString());
    }
    public Buffer()
    {
        messageBuffer = new Queue();
    }
}

class Messenger
{
    static void Main(string[] args)
    {
        Buffer objBuffer = new Buffer();
        objBuffer.SendMessage(new Message("One"));
        objBuffer.SendMessage(new Message("Two"));
        objBuffer.ReceiveMessage();
        objBuffer.SendMessage(new Message("Three"));
        objBuffer.ReceiveMessage();
        objBuffer.SendMessage(new Message("Four"));
        objBuffer.ReceiveMessage();
        objBuffer.ReceiveMessage();

        Console.ReadKey();
    }
}
```

Example: Stack

```
class Message
```

```
{
    private string messageText;
    public Message(string str)
    {
        messageText = str;
    }
    public override string ToString()
    {
        return messageText;
    }
}

class Buffer
{
    private Stack messageBuffer;
    public void SendMessage(Message msg)
    {
        messageBuffer.Push(msg);
    }
    public void ReceiveMessage()
    {
        Message message = (Message)messageBuffer.Pop();
        Console.WriteLine(message.ToString());
    }
    public Buffer()
    {
        messageBuffer = new Stack();
    }
}

class Messenger
{
    static void Main(string[] args)
    {
        Buffer buffer = new Buffer();
        buffer.SendMessage(new Message("One"));
        buffer.SendMessage(new Message("Two"));
        buffer.ReceiveMessage();
        buffer.SendMessage(new Message("Three"));
        buffer.ReceiveMessage();
        buffer.SendMessage(new Message("Four"));
        buffer.ReceiveMessage();
        buffer.ReceiveMessage();

        Console.ReadKey();
    }
}
```


Use Hash Tables

- A hash table is a data structure that is **designed** for **fast retrieval**
- It does this by **associating a key with each object** that you store in the table.

Example:

```
// A library contains a list of books.
class Library
{
    public Hashtable bookList;
    public Library()
    {
        bookList = new Hashtable();
    }
}

// Books are placed in the library
class Book
{
    public string Title;
    public int ISBN;

    public Book(string title, int isbn)
    {
        Title = title; ISBN = isbn;
    }
}

class ClassMain
{
    static void Main(string[] args)
    {
        Book book1 = new Book("Programming with C#", 0735613702);
        Book book2 = new Book("Inside C#", 0735612889);
        Book book3 = new Book("Microsoft C# Language Specifications", 0735614482);
        Book book4 = new Book("OOP with C#", 0735615543);

        Library myReferences = new Library();
        myReferences.bookList.Add(book1.ISBN, book1);
        myReferences.bookList.Add(book2.ISBN, book2);
        myReferences.bookList.Add(book3.ISBN, book3);
        myReferences.bookList.Add(book4.ISBN, book4);

        Book book = (Book)myReferences.bookList[0735612889];

        Console.WriteLine(book.Title);
        Console.ReadKey();
    }
}
```


Chapter 12: Interfaces

What is Interfaces?

- Interface is a **reference type** that defines **contact**
- In other words, you can say it's as object **access point**
- Can **contain** method, properties, indexers, event.
- Does **not** provides **implementation** for the member
- Can **inherits** zero or more Interface
- An interface contains only the signatures of methods, properties, events or indexers.

- A class or struct that implements the interface must implement the members of the interface
- One interface can inherit another interface but cannot implement method.

Purposes of Interface

- Interfaces are better **suited** to situations in which your applications require **many possibly unrelated object types to provide certain functionality**.
- Interfaces permit **polymorphism** between classes with different base classes.
- Interfaces are more flexible than base classes because you can define **a single implementation that can implement multiple interfaces.**
- Interfaces are better in situations in which you **do not need to inherit implementation** from a base class.
- Interfaces are useful in cases where you **cannot use class inheritance**

Example: Interface

```
interface ICarnivore
{
    bool IsHungry { get; }
    void Hunt();
    void Eat(string victim);
}

public class Lion : ICarnivore
{
    private bool hungry;
    public bool IsHungry
    {
        get
        {
            return hungry;
        }
    }

    public void Hunt()
    {
        Console.WriteLine("Lion hunt");
    }

    public void Eat(string victim)
    {
        Console.WriteLine("Lion eats" + victim);

    }
}


class Program
{
    static void Main(string[] args)
    {
        ICarnivore objICarnivoure = new Lion();
```

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

Example 2:

```
   interface ICarnivore
    {
        bool IsHungry { get; }
        Animal Hunt();
        void Eat(Animal victim);
    }

    public abstract class Animal
    {
        public abstract void Sleep();
    }

    public class Antelope : Animal
    {
        public override void Sleep() { }
    }

    public class Lion : Animal, ICarnivore
    {

        public Lion()
        {
            hungry = true;
        }

        // ICarnivore implementation
        private bool hungry;
        public bool IsHungry
        {
            get
            {
                return hungry;
            }
        }

        public Animal Hunt()
        {
            // hunt and capture implementation
            return new Antelope();
        }

        public void Eat(Animal prey)
        {
            // implementation
            Console.WriteLine("Lion is no longer hungry");
        }

        // Inherited from base class
        public override void Sleep()
```

```
        {
            // sleeping
        }

        public void JoinPride()
        {
            // Join with a Pride of other Lions
        }
    }


    class Tester
    {
        static void Main(string[] args)
        {
            Lion aLion = new Lion();
            Antelope a = new Antelope();

            // carnivore-like behavior
            if (aLion.IsHungry)
            {
                Animal victim = aLion.Hunt();
                if (victim != null)
                {
                    aLion.Eat(victim);
                }
            }

            // Lion specific
            aLion.JoinPride();
            // Animal behavior
            aLion.Sleep();
            Console.ReadKey();
        }
    }
```

Work with Objects that Implement Interfaces

- Can use **the *is* and *as* keywords** to determine whether an object **implements** a specific interface

```
public abstract class Animal
{
    public abstract void Sleep();
}

interface ICarnivore
{
    bool IsHungry { get; }
    Animal Hunt();
    void Eat(Animal victim);
}
interface IHerbivore
{
    bool Hungry { get; }
```

```
        void GatherFood();
    }

    public class Chimpanzee : Animal, IHerbivore, ICarnivore
    {
        // implement members of IHerbivore and ICarnivore
        public override void Sleep()
        {
            Console.WriteLine("Chimpanzee is sleeping.");
        }

        public bool Hungry
        {
            get { throw new NotImplementedException(); }
        }

        public void GatherFood()
        {
            Console.WriteLine("Do you know, how Chimpanzee gather its food?");
        }

        public bool IsHungry
        {
            get { throw new NotImplementedException(); }
        }

        public Animal Hunt()
        {
            throw new NotImplementedException();
        }

        public void Eat(Animal victim)
        {
            throw new NotImplementedException();
        }
    }

    public class Lion : Animal, ICarnivore
    {
        public void Eat(Animal prey)
        {
            // implementation
            if (prey is Antelope)
            {
                Console.WriteLine("Eat Antelope's favorite meal");
            }
            else
            {
              Console.WriteLine("Animal is no longer hungry");
            }
        }


        public override void Sleep()
        {
            Console.WriteLine("Lion is sleeping.");
        }
```

```
        public bool IsHungry
        {
            get { throw new NotImplementedException(); }
        }

        public Animal Hunt()
        {
            throw new NotImplementedException();
        }
    }

    public class Antelope : Animal, IHerbivore
    {
        // implement members of IHerbivore
        public override void Sleep()
        {
            Console.WriteLine("Lion is sleeping.");
        }

        public bool Hungry
        {
            get { throw new NotImplementedException(); }
        }

        public void GatherFood()
        {
            Console.WriteLine("Do you know, how Antelope gather its food?");
        }
    }

    public class elephant : Animal, IHerbivore
    {
        // implement members of IHerbivore
        public override void Sleep()
        {
            Console.WriteLine("Elepant is sleeping.");
        }

        public bool Hungry
        {
            get { throw new NotImplementedException(); }
        }

        public void GatherFood()
        {
            Console.WriteLine("Do you know, how elephant gather its food?");
        }
    }



    class Program
    {
        static void Main(string[] args)
        {

            List<Animal> zoo = new List<Animal>();
```

```
            Animal chipanzee = new Chimpanzee();
            zoo.Add(chipanzee);

            Animal lion = new Lion();
            zoo.Add(lion);

            Animal antelope = new Antelope();
            zoo.Add(antelope);

            Animal elephant = new elephant();
            zoo.Add(elephant);

            foreach (Animal someAnimal in zoo)
            {

                if (someAnimal is IHerbivore)
                {
                    //IHerbivore veggie = (IHerbivore)someAnimal;
                    IHerbivore veggie = someAnimal as IHerbivore;
                    veggie.GatherFood();
                }
                else if (someAnimal is ICarnivore)
                {
                    ICarnivore carnivore = someAnimal as ICarnivore;
                    //carnivore.Hunt();
                }
            }

            Console.ReadKey();

        }
    }
```

Multiple Interfaces in C#

- A **class** can inherit **multiple interfaces.**
- Interfaces can also inherit from one or more interfaces.
- To implement multiple interface inheritance, list the interfaces in a **comma-separated list**

- Example:

```
public class Chimpanzee : Animal, IHerbivore, ICarnivore

public abstract class Animal
{
    public abstract void Sleep();
}


interface ICarnivore
{
    bool IsHungry { get; }
    Animal Hunt();
```

```
        void Eat(Animal victim);
    }
    interface IHerbivore
    {
        bool IsHungry { get; }
        void GatherFood();
    }

    interface IOmnivore : IHerbivore, ICarnivore
    {
        void DecideWhereToGoForDinner();
    }

    public class Chimpanzee : Animal, IOmnivore
    {
        bool ICarnivore.IsHungry
        {
            get
            {
                return false;
            }
        }

        bool IHerbivore.IsHungry
        {
            get
            {
                return true;
            }
        }

        public void DecideWhereToGoForDinner()
        {
            throw new NotImplementedException();
        }


        public void GatherFood()
        {
            throw new NotImplementedException();
        }


        public Animal Hunt()
        {
            throw new NotImplementedException();
        }

        public void Eat(Animal victim)
        {
            throw new NotImplementedException();
        }

        public override void Sleep()
        {
            throw new NotImplementedException();
        }
    }
```

```
class Program
{
    static void Main(string[] args)
    {
        IOmnivore objIOmnivore = new Chimpanzee();

        Console.WriteLine(((ICarnivore)objIOmnivore).IsHungry);
        Console.WriteLine(((IHerbivore)objIOmnivore).IsHungry);

        Console.ReadKey();
    }
}
```

Interfaces vs. abstract classes

- **Abstract Class**
  - Is a **special kind of class** that **cannot be instantiated**
  - An abstract class is only to be **sub-classed**(inherited from)
  - The advantage is that it **enforces certain hierarchies** for all the subclasses
  - It is a kind of **contract that forces all the subclasses to carry on the same hierarchies or standards**

- **Interface**
  - An **interface is not a class**
  - **It is an entity** that is defined by the **word Interface**
  - Interface has **no implementation**; it only has the **signature**
  - The main difference between them is that a class can implement **more than one interface** but **can only inherit from one abstract class**
  - Since C# doesn't support multiple inheritance, interfaces are used to implement **multiple inheritance.**

Chapter 13: Exception Handling

What is Exception Handling?

- An exception is any **error condition** or **unexpected behavior** that is encountered by an executing program

- Exceptions can be raised because of
  - **a fault in your code** or in code that you call
  - **resources not being available**, such as running out of memory
  - not being able to **locate a particular file**
  - Other **unexpected conditions.**

- You can handle these error conditions by using **try**, **catch**, and **finally** keywords.

The following examples represent situations that might cause exceptions:

Scenario	Solution
Opening a file	Before opening a file, check that the file **exists** and that you can open it.
Reading an XML document	You will normally read **well-formed XML documents**, but you should deal with the exceptional case where the document is **not valid XML**. This is a good example of where to rely on exception handling.
Accessing an invalid member of an array	If you are the user of the array, this is an application bug that you should eliminate. Therefore, you should not use exception handling to catch this exception.
Dividing a number by zero	This can normally be checked and avoided.
Converting between types using the **System.Convert** classes	With some checking, these can be avoided.

```
try
{
    byte tickets = Convert.ToByte(numberOfTickets.Text);
}
catch (FormatException)
{
    MessageBox.Show("Format Exception: please enter a number");
}
```

Handling specific exception

```
try
{
    byte tickets = Convert.ToByte(numberOfTickets.Text);
}
catch (FormatException)
{
    MessageBox.Show("Format Exception: please enter a number");
}
```

Multiple catch blocks

```
try
{
    byte tickets = Convert.ToByte(numberOfTickets.Text);
}
catch (FormatException e)
```

```
{
    MessageBox.Show("Format Exception: please enter a number");
}
catch (OverflowException e) // Byte range : 0 to 255
{
    MessageBox.Show("Overflow: too many tickets");
}
```

Using the finally keyword

- A **finally** block is always executed, regardless of whether an exception is thrown.

```
FileStream xmlFile = null;
try
{
    xmlFile = new FileStream("XmlFile.xml", FileMode.Open);
}
catch (System.IO.IOException e)
{
    return;
}
finally
{
    if (xmlFile != null)
    {
        xmlFile.Close();
    }
}
```

Throwing user-defined exceptions

- You can create your own exception classes by deriving from the **Application.Exception** class

Format Exception

```
private int ReadData()
{
    byte tickets = 0;
    try
    {
        tickets = Convert.ToByte(textBox1.Text);
    }
    catch (FormatException e)
    {
        if (textBox1.Text.Length == 0)
        {
            throw (new FormatException("No user input ", e));
        }
        else
        {
```

```
                throw e;
            }
        }
        return tickets;
    }
```

Calling Exception

```
        int tickets = 0;
        try
        {
            tickets = ReadData();
        }
        catch (FormatException e)
        {
            MessageBox.Show(e.Message + "\n");
            MessageBox.Show(e.InnerException.Message);
        }
```

Example 2:

```
class TicketException : ApplicationException
{
    private bool purchasedCompleted = false;
    public bool PurchaseWasCompleted
    {
        get
        {
            return purchasedCompleted;
        }
    }
    public TicketException(bool completed, Exception e)
        : base("Ticket Purchase Error", e)
    {
        purchasedCompleted = completed;
    }
}

class Program
{
    public static string input = string.Empty;
    static void Main(string[] args)
    {
        input = "5000";
        int tickets = 0;
        try
        {
            tickets = ReadData();
        }
        catch (TicketException e)
        {
            Console.WriteLine(e.Message);
            Console.WriteLine(e.InnerException.Message);
        }
```

```
            Console.ReadKey();
        }

        private static int ReadData()
        {
            byte tickets = 0;
            try
            {
                tickets = Convert.ToByte(input);
            }
            catch (Exception e)
            {
                // check if purchase was complete
                throw (new TicketException(true, e));
            }
            return tickets;
        }
    }
```

Chapter 14: Generics

Introduction to Generics

- Generics are the most powerful feature **of C# 2.0**
- It is allow you to **type-safe data** structure
- Improve **performance** and **quality** code
- Similar concept of **C++ template** but implementation and capability is different

What are generics?

- Consider **Stack** in C# 1.1, you are using **object based Push() and Pop() method**
    - Significant performance overhead for **type casting**
- Generics **implements in once** and you can implement and you can declare and **implement it for any type**
- To do that, use the **< and >** brackets, enclosing a generic type parameter.

Example:

```
public class Stack<T>
{
    T[] m_Items;
    public void Push(T item)
    {
        //.....
    }
    public T Pop()
    {
        //.....
    }
```

```
}

Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```

Benefits of Generics

- Generics in .NET let you **reuse code**
- You can **develop, test, and deploy** your **code once**, reuse it with **any type, including future types,** all with full compiler support and type safety.
- Because the generic code does not force the **boxing and unboxing** of value types, or the down casting of reference types, **performance is greatly** improved
- With **value types** there is typically a **200 percent performance gain**, and with **reference types** you can expect up to a **100 percent performance gain** in accessing the type

Applying Generics

Example:

```
public struct Point<T>
{

    public T X;
    public T Y;
}


class Program
{
    static void Main(string[] args)
    {
        Point<int> point;
        point.X = 3;
        point.Y = 4;

        Console.WriteLine(point.X + "," + point.Y);

        Point<double> dpoint;
        dpoint.X = 3.5;
        dpoint.Y = 6.5;

        Console.WriteLine(dpoint.X + "," + dpoint.Y);
        Console.ReadKey();
    }
}
```

- If you were using an **Object-based stack, you would simply return null**, but a generic stack could be used with value types as well. To address this issue, you can use the **default()**

```
public T Pop()
{
   m_StackPointer--;
   if(m_StackPointer >= 0)
   {
      return m_Items[m_StackPointer];
   }
   else
   {
      m_StackPointer = 0;
      return default(T);
   }
}
```

Multiple Generic Types

- A single type can define multiple generic-type parameters.

```
class Node<K, T>
{
    public K Key;
    public T Item;
    public Node<K, T> NextNode;
    public Node()
    {
        Key = default(K);
        Item = default(T);
        NextNode = null;
    }
    public Node(K key, T item, Node<K, T> nextNode)
    {
        Key = key;
        Item = item;
        NextNode = nextNode;
    }
}

public class LinkedList<K, T>
{
    Node<K, T> m_Head;
    public LinkedList()
    {
        m_Head = new Node<K, T>();
    }

    public void AddHead(K key, T item)
    {
        Node<K, T> newNode = new Node<K, T>(key, item, m_Head.NextNode);
        m_Head.NextNode = newNode;
    }
}


class Program
{
    static void Main(string[] args)
```

```
    {
        LinkedList<int, string> list = new LinkedList<int, string>();
        list.AddHead(1, "A");
        list.AddHead(2, "B");
        list.AddHead(3, "C");
        list.AddHead(4, "D");
        list.AddHead(5, "E");
        Console.ReadKey();
    }
}
```

Generic Constraints

- In C# you need to instruct the compiler **which constraints the client-specified types must obey** in order for them to be used instead of the generic type parameters.
- There are three types of constraints.
    - **A derivation constraint** indicates to the compiler that the **generic type parameter derives from a base type** such an interface or a particular base class.
    - **A default constructor constraint** indicates to the compiler that **the generic type parameter exposes a default public constructor** (a public constructor with no parameters).
    - **A reference/value type constraint** constrains the generic type parameter **to be a reference or a value type.** A generic type can employ multiple constraints, and you even get IntelliSense reflecting the constraints when using the generic type parameter
- Although **constraints are optional, they are often essential** when developing a generic type

Derivation Constraints

- Use the **where** keyword on the generic type parameter followed by a **derivation colon** to indicate to the compiler that the generic type parameter implements a particular interface.

```
class Node<K, T>
{
    public K Key;
    public T Item;
    public Node<K, T> NextNode;
    public Node()
    {
        Key = default(K);
        Item = default(T);
        NextNode = null;
    }
    public Node(K key, T item, Node<K, T> nextNode)
    {
        Key = key;
        Item = item;
        NextNode = nextNode;
    }
```

```
}

public class LinkedList<K, T> where K : IComparable
{
    Node<K, T> m_Head;
    public LinkedList()
    {
        m_Head = new Node<K, T>();
    }

    public void AddHead(K key, T item)
    {
        Node<K, T> newNode = new Node<K, T>(key, item, m_Head.NextNode);
        m_Head.NextNode = newNode;
    }

    public T Find(K key)
    {
        Node<K, T> current = m_Head;
        while (current.NextNode != null)
        {
            if (current.Key.CompareTo(key) == 0)

                break;
            else

                current = current.NextNode;
        }
        return current.Item;
    }
}


class Program
{
    static void Main(string[] args)
    {
        LinkedList<int, string> list = new LinkedList<int, string>();
        list.AddHead(1, "A");
        list.AddHead(2, "B");
        list.AddHead(3, "C");
        list.AddHead(4, "D");
        list.AddHead(5, "E");

        string value = list.Find(2);
        Console.WriteLine(value);

        Console.ReadKey();
    }
}
```

- Note that even though the constraint allows you to use **IComparable**, it does not eliminate the boxing penalty when the key used is a value type, such as an integer. To overcome this, the System.Collections.Generic namespace defines the generic interface **IComparable<T>**:

```
public interface IComparable<T>
{
   int CompareTo(T other);
   bool Equals(T other);
}
```

- You can constrain the key type parameter to support **IComparable<T>** with the key's type as the type parameter, and by doing so you gain not only type safety but also eliminate the boxing of value types when used as keys:

```
public class LinkedList<K,T> where K : IComparable<K>
{...}
```

- In C# 2.0, all constraints must appear after the actual derivation list of the generic class.

```
public class LinkedList<K,T> : IEnumerable<T> where K : IComparable<K>
{...}
```

Constructor Constraint

- Suppose you want to instantiate a **new generic object inside a generic class**. The problem is the **C# compiler does not know whether the type argument the client will use has a matching constructor**, and it will **refuse to compile the instantiation** line.

- To address this problem, C# allows you to constrain a generic type parameter such **that it must support a public default constructor.** This is done using the **new()** constraint.

```
class Node<K,T> where T : new()
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;
   public Node()
   {
      Key      = default(K);
      Item     = new T();
      NextNode = null;
   }
}
```

Reference/Value Type Constraint

- You can **constrain a generic type parameter to be a value type** (such as an int, a bool, and enum) or any custom structure using the **struct** constraint:

```
public class MyClass<T> where T : struct

{...}
```

- Similarly, you **can constrain a generic type parameter to be a reference type** (a class) using the **class** constraint:

```
public class MyClass<T> where T : class

{...}
```

Generic Methods

Example:

```
public class MyClass
{
   public void MyMethod<T>(T t)
   {...}
}


MyClass obj = new MyClass();
obj.MyMethod<int>(3);
```

Example:

```
public class MyClass
{
   public void SomeMethod<T>(T t) where T : IComparable<T>
   {...}
}
```

- The **subclass implementation** must repeat all constraints that appeared at the base method level:

```
public class BaseClass
{
   public virtual void SomeMethod<T>(T t) where T : new()
   {...}
}
public class SubClass : BaseClass
{
   public override void SomeMethod<T>(T t) where T : new()
   {...}
}
```

Chapter 15: ADO.NET

What is ADO.NET?

ADO.NET is a **data access technology**.

- ADO Stands for **Active Data Object**
- A set of **classes, interfaces, structures, and enumerations** that **manage data access** from within the .NET Framework.
- A system designed for **disconnected environments**.
- A programming model with advanced **XML support.**

Benefit of ADO.NET

- **Scalability**
  - The ADO.NET programming model encourages programmers to **conserve system resources** for applications that run on the **Web**.
  - Because data is held **locally in in-memory caches**, there is **no need to maintain active database connections** for extended periods.

- **Programmability**
  - The ADO.NET programming model **uses strongly typed data.**
  - **Strongly typed data makes code more concise and easier** to write

- **Interoperability.**
  - ADO.NET makes **extensive use of XML.**
  - XML is a portable, text-based technology to represent data in an **open and platform-independent way,** which makes it easier to pass data between applications even if they are running on different platforms.

ADO.NET Components

- The ADO.NET **components** are designed to **separate data access** from **data manipulation**.
- The **two components of ADO.NET** that accomplish this
  - are the **DataSet** object and the **.NET data provider.**
- The components of the **.NET data provider** are explicitly designed for **disconnected data manipulations.**

What is connected Environment?

- A connected environment is one in which a user or an **application** is continuously connected to a **data source.**

Advantages

- A **secure** environment is easier to maintain.
- **Concurrency** is easier to control.
- Data is more likely to be **current than** in other scenarios.

Disadvantages

- It must have a constant database connection.
- It is **not scalable**.

Example of connected environment

- A factory that requires a real-time connection to monitor production output and storage.
- A brokerage house that requires a constant connection to stock quotes.

What is disconnected environment?

- A disconnected environment is one in which a **user or an application** is **not constantly connected** to a **source of data**.
- **Mobile users who work with laptop** computers are the primary users in a disconnected environment.
- Users can take a subset of data with them on a disconnected computer and then merge changes back into the central data store.

Advantages

- You can **work at any time** that is convenient for you, and you can connect to a data source at any time to process requests.
- Others can share connection resources.
- The **scalability** and **performance** of applications is **improved**.

Example: When you return your rental car, the person who accepts the car uses a handheld computer to read the return information. Because the handheld device may have limited processing capacity, it is important to scale the data to the task that the user performs at any given time

Disadvantages

- Data is **not** always **up to date**.
- **Change conflicts** can occur and must be resolved.

What is the ADO.NET Object Model?

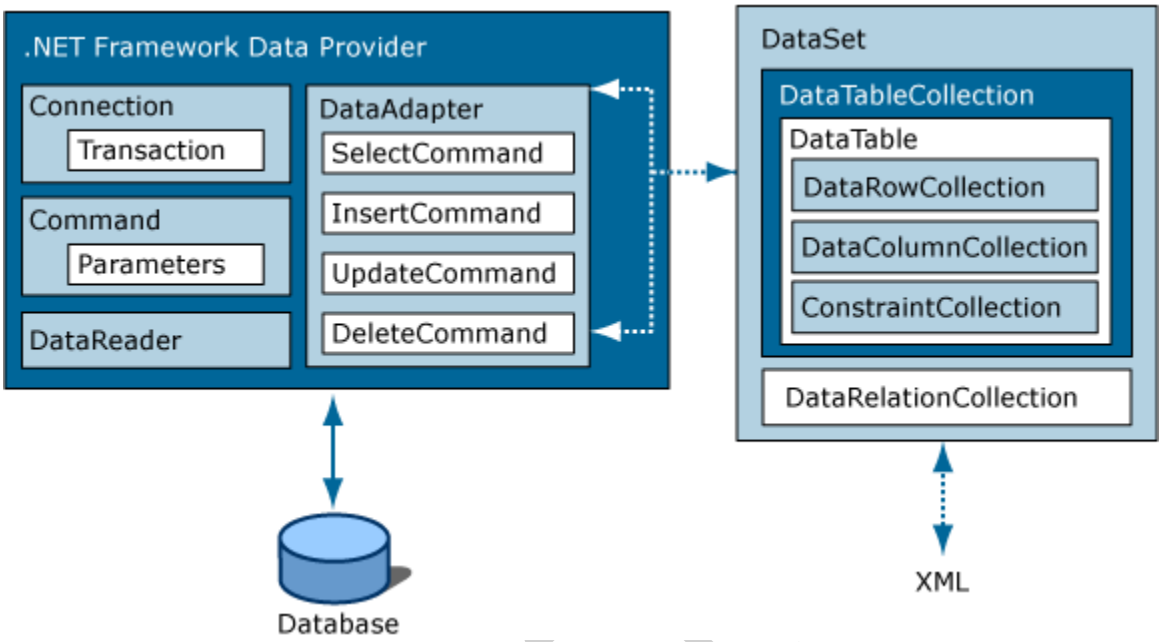
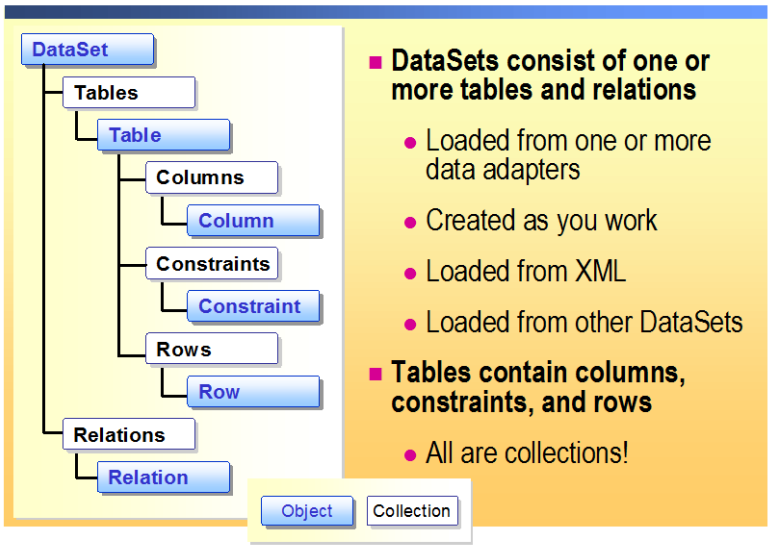


Fig – ADO.NET Object Model

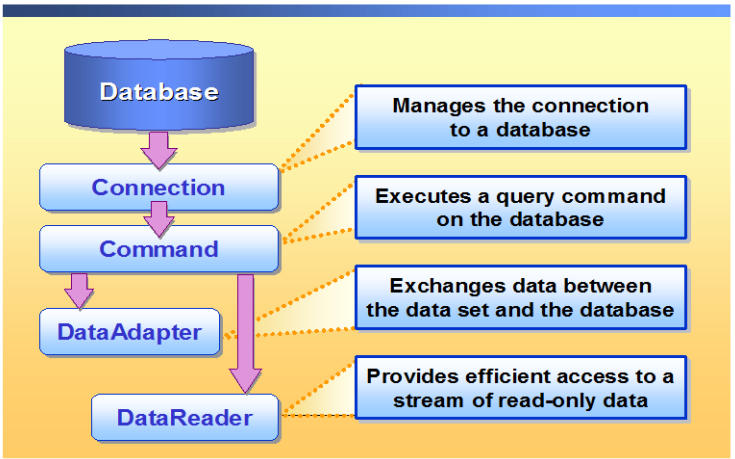
- The ADO.NET object model consists of two major parts:
    - **.NET data provider** classes
        - Classes are **specific to a data** source
            - SQL Server .NET Data Provider
            - OLE DB .NET Data Provider
    - **DataSet** class
        - The **DataSet** class allows you to **store and manage data in a disconnected cache**
        - The **DataSet** is **independent of any underlying data source**, so its features are available to all applications, regardless of the origin of the data in the application.

What is dataset Class?



- The ADO.NET **DataSet** class is the core component of the disconnected architecture of ADO.NET
- The **DataSet** class contains collections of:
    - ***Tables***. Tables are stored as a collection of **DataTable** objects, which in turn each hold collections of **DataColumn** and **DataRow** objects.
    - ***Relations***. Relations are stored as a collection of **DataRelation** objects that describe the relationships between tables.
    - ***Constraints***. Constraints track the information that **ensures data integrity**

What is the .NET Data Provider?



- A .NET data provider enables you to **connect to a database, execute commands, and retrieve results.**
- Those results are **either processed directly** or placed in an ADO.NET **DataSet** object to be exposed to the user in an ad-hoc manner, combined with data from multiple sources, or used remotely.
- A .NET data provider can handle **basic data manipulation**, such as updates, inserts, and basic data processing.
- Its primary focus is to **retrieve data from a data source and pass it on to a DataSet object**, where your application can use it in a disconnected environment.
- ADO.NET provides two kinds of .NET data providers:
    - **SQL Server .NET Data Provider**
        - The SQL Server .NET Data Provider accesses databases in **Microsoft SQL Server version 7.0 or later**.
        - It provides excellent performance because it accesses SQL Server directly instead of going through an intermediate OLE DB provider.
    - **OLE DB .NET Data Provider**
        - The OLE DB .NET Data Provider accesses databases **in SQL Server 6.5 or earlier, Oracle, and Microsoft Access.**

.NET data provider classes

Class	Description
Connection	**Establishes and manages a connection** to a **specific data source**. For example, the **SqlConnection** class connects to OLE DB data sources.
Command	**Executes a query command** from a data source. For example, the **SqlCommand** class can execute SQL statements in an OLE DB data source.
DataAdapter	Uses the **Connection**, **Command**, and **DataReader** classes implicitly **to populate a DataSet** object and to update the central data source with any changes made to the **DataSet**. For example, the **SqlDataAdapter** object can manage the interaction between a **DataSet** and an Access database.
DataReader	Provides an efficient, **forward-only, read-only** stream of data from a data source.

How to Specify the Database Connection

- Use the connection object to connect to a specific data source

- Before you can work with data, you must first establish a connection to a data source
  - use the **Connection** object to connect to a specific data source
- Choose the connection type
  - You can use either the **SqlConnection** object to connect to a SQL Server database
  - The **OleDbConnection** object to connect to other types of data sources.
- Specify the data source
- Open the connection to the data source
- Use a **ConnectionString** property to specify the data provider, the data source, and other information that is used to establish the connection.

- Use the **connection string** to specify all the options for your connection to the database, including the **account name, database server, and database name**

Parameter of ConnectionString

Parameter	Description
Initial Catalog	The name of the database.
Data Source	The name of the SQL Server to be used when a connection is open, or the filename of a Microsoft Access database.
Integrated Security or Trusted Connection	The parameter that determines whether **the connection is to be a secure connection. True**, **False**, and **SSPI** are the possible values. **SSPI** is the equivalent of **True**.
User ID	The SQL Server login account.
Password	The login password for the SQL Server account.
Provider	The property used to set or return the name of the provider for the connection, used only for **OleDbConnection** objects.
Connection Timeout or Connect Timeout	The length of time in seconds to wait for a connection to the server before terminating the attempt and generating an exception. **30** is the default.
Persist Security Info	When set to **False**, security-sensitive information, such as the password, is not returned as part of the connection if the connection is open or has ever been in an open state. Setting this property to **True** can be a security risk. **False** is the default.

Example of ConnectionString

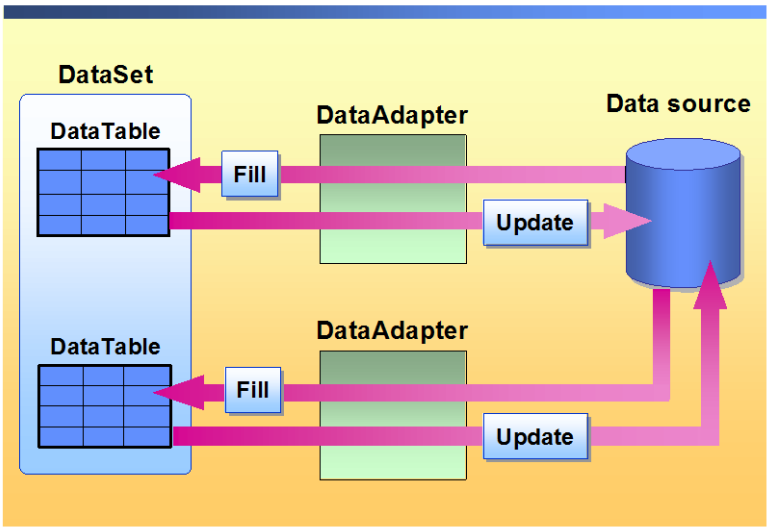
```
string connectionString = @"data source=localhost;integrated security=SSPI;initial catalog=Northwind";

string connectionString = @"provider=Microsoft.JET.OLEDB.4.0;data
source=C:\samples\northwind.mdb";
```

How to Specify the Database Command

```
string commandString = @"SELECT CompanyName, ContactName FROM Customers WHERE
CompanyName='Island Trading'";
```

- Create a string containing SQL Statements
  - Remember that **verbatim string can** make this much easier!
- Example of SQL Statements
  - string queryString = "SELECT * FROM Customers";
  - string commandString = @"SELECT * FROM Customers WHERE CompanyName='Island Trading'";
  - string commandString = @"SELECT CompanyName, ContactName FROM Customers WHERE CompanyName='Island Trading'";
  - string commandString = @"**SELECT** Products.ProductID, Products.ProductName, Products.SupplierID, Products.CategoryID, Products.QuantityPerUnit, Products.UnitPrice, Suppliers.CompanyName, Suppliers.SupplierID **AS** Expr1 **FROM** Products **INNER JOIN** Suppliers **ON** Products.SupplierID = Suppliers.SupplierID";

How to Create the DataAdapter Object



- To establish the connection to the data source and manage the movement of data to and from the data source, you use a **DataAdapter** object.
- A **DataAdapter** object serves as a **bridge** between a **DataSet** object and a data source

- The **DataAdapter** object represents a set of database commands and a database connection that you use to fill a **DataSet** object and update the data source.
- **DataAdapter** objects are **part of the .NET data providers**, which also **include connection objects, data-reader objects, and command objects.**
- Each **DataAdapter** object exchanges data between a single **DataTable** object in a dataset and a single result set from a SQL statement.
- Use one **DataAdapter** object for **each query** when you send more than one query to a database from your application.

DataAdapter properties

Select Command	Retrieves rows from the data source.
InsertCommand	Writes inserted rows from the DataSet into the data source.
UpdateCommand	Writes modified rows from the DataSet into the data source.
DeleteCommand	Deletes rows in the data source.

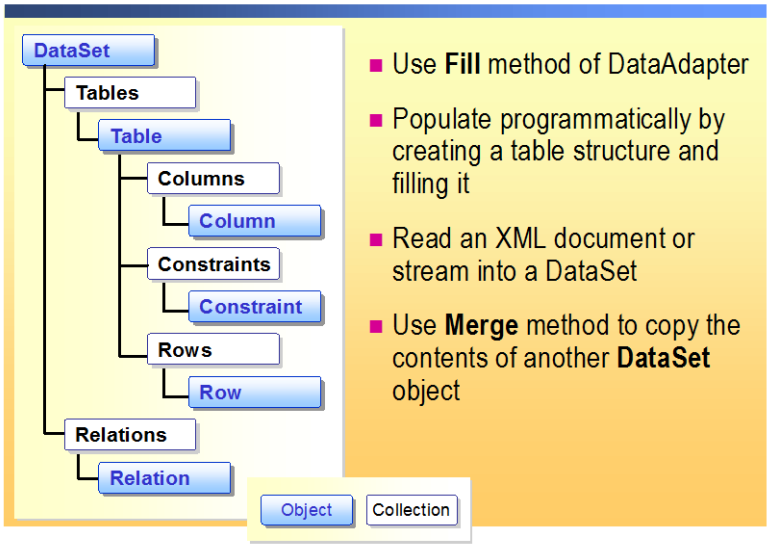
Uses of DataAdapter

```
    class Program
    {
        static void Main(string[] args)
        {
  // string connectionString = @"data source=(localDB)\v11.0; Initial catalog=TrainingDB;
integrated security=SSPI";

 //string connectionString = "data source=192.168.20.125;database=SMSDB;Integrated
Security=false; user id=sa; password=leads@123";

string connectionString=@"Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\TrainingDB.mdf;Initial
Catalog=TrainingDB;Integrated Security=True";

            string commandString = @"SELECT * FROM Employee";
            SqlDataAdapter dataAdapter = new SqlDataAdapter(commandString,
connectionString);

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill(myDataSet);

            DataTable table = myDataSet.Tables[0];
            int numberRows = table.Rows.Count;

            Console.WriteLine(numberRows);
            Console.ReadKey();

        }
```

Example of **OleDbDataAdapter**

```
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = @"provider=Microsoft.JET.OLEDB.4.0;data
source=c:\samples\Northwind.mdb";
            string commandString = @"SELECT * FROM Customers";
            OleDbDataAdapter dataAdapter = new OleDbDataAdapter(commandString,
connectionString);

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill(myDataSet);

            DataTable table = myDataSet.Tables[0];
            int numberRows = table.Rows.Count;
            Console.WriteLine(numberRows);
            Console.ReadKey();
        }
    }
```

How to create a DataSet Object



- **DataSet** objects store data in a disconnected cache.
- A **DataSet** is a container; therefore, you must populate it with data. You can populate a **DataSet** in a variety of ways:
    - Call the **Fill** method of a data adapter
    - Manually populate tables in the **DataSet**
    - Read an XML document or stream into the **DataSet**
    - Copy or merge the contents of another **DataSet**

Example 1:

```
    class Program
    {
        static void Main(string[] args)
        {
            // string connectionString = @"data source=(localDB)\v11.0; Initial
catalog=TrainingDB; integrated security=SSPI";

            //string connectionString = "data
source=192.168.20.125;database=SMSDB;Integrated Security=false; user id=sa;
password=leads@123";

            string connectionString = @"Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\TrainingDB.mdf;Initial
Catalog=TrainingDB;Integrated Security=True";

            string commandString = @"SELECT * FROM Employee";
            SqlDataAdapter dataAdapter = new SqlDataAdapter(commandString,
connectionString);

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill(myDataSet);

            DataTable table = myDataSet.Tables[0];
            int numberRows = table.Rows.Count;

            Console.WriteLine(numberRows);
            Console.ReadKey();


        }
    }
```

Example 2:

```
    class Program
    {
        static void Main(string[] args)
        {
            // string connectionString = @"data source=(localDB)\v11.0; Initial
catalog=TrainingDB; integrated security=SSPI";

            //string connectionString = "data
source=192.168.20.125;database=SMSDB;Integrated Security=false; user id=sa;
password=leads@123";

            string connectionString = @"Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\TrainingDB.mdf;Initial
Catalog=TrainingDB;Integrated Security=True";

            string commandString = @"SELECT * FROM Employee";
            SqlDataAdapter dataAdapter = new SqlDataAdapter(commandString,
connectionString);
```

```
            DataSet myDataSet = new DataSet();
            dataAdapter.Fill(myDataSet, "tbl_employee");

            DataTable table = myDataSet.Tables["tbl_employee"];

            foreach (DataRow row in table.Rows) // Loop over the rows.
            {
                Console.WriteLine("--- Row ---"); // Print separator.
                foreach (var item in row.ItemArray) // Loop over the items.
                {
                    Console.Write("Item: "); // Print label.
                    Console.WriteLine(item);
                }
            }
            Console.ReadKey();
        }
    }
```

Example 3:

```
    class Program
    {
        static void Main(string[] args)
        {
            // string connectionString = @"data source=(localDB)\v11.0; Initial
catalog=TrainingDB; integrated security=SSPI";

            //string connectionString = "data
source=192.168.20.125;database=SMSDB;Integrated Security=false; user id=sa;
password=leads@123";

            string connectionString = @"Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\TrainingDB.mdf;Initial
Catalog=TrainingDB;Integrated Security=True";

            string commandString = @"SELECT * FROM Employee";
            SqlDataAdapter dataAdapter = new SqlDataAdapter(commandString,
connectionString);

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill(myDataSet, "tbl_Employee");

            DataTable table = myDataSet.Tables["tbl_Employee"];

            int numberRows = table.Rows.Count;
            Console.WriteLine("Rows: {0} ", numberRows);

            DataColumn c = table.Columns[0];
            Console.WriteLine("Column one: {0}", c.ColumnName);
            Console.WriteLine("== Full Name ==");

            foreach (DataRow row in table.Rows) // Loop over the rows.
            {
                Console.WriteLine(row[1]);
            }
```

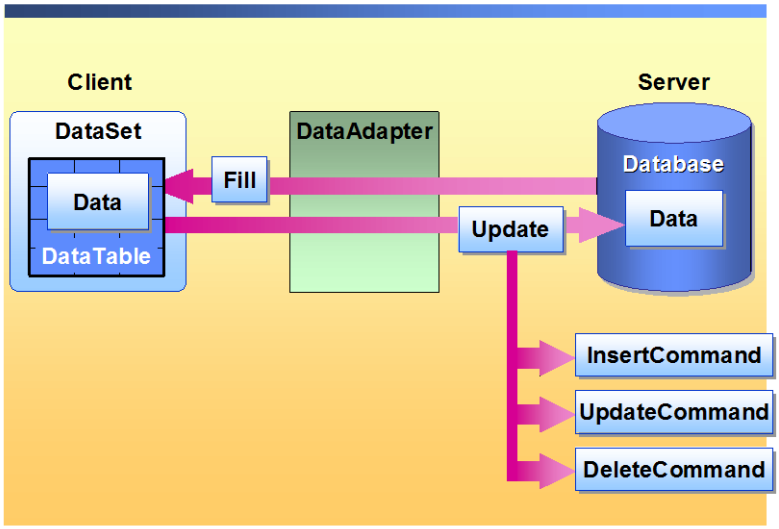
```
            Console.ReadKey();

        }
    }
```

Example 4:

```
    class Program
    {
        static void Main(string[] args)
        {
            // string connectionString = @"data source=(localDB)\v11.0; Initial
catalog=TrainingDB; integrated security=SSPI";

            //string connectionString = "data
source=192.168.20.125;database=SMSDB;Integrated Security=false; user id=sa;
password=leads@123";

            string connectionString = @"Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\TrainingDB.mdf;Initial
Catalog=TrainingDB;Integrated Security=True";

            string commandString = @"SELECT * FROM Suppliers";

            SqlDataAdapter dataAdapter =
                    new SqlDataAdapter(commandString, connectionString);


            commandString = @"SELECT * FROM Products";
            SqlDataAdapter dataAdapter2 =
                    new SqlDataAdapter(commandString, connectionString);

            DataSet myDataSet = new DataSet();

            dataAdapter.Fill(myDataSet, "tbl_suppliers");
            dataAdapter2.Fill(myDataSet, "tbl_products");

            int tableCount = myDataSet.Tables.Count;

            DataRelation dr = myDataSet.Relations.Add("ProductSuppliers",
                        myDataSet.Tables["tbl_suppliers"].Columns["SupplierID"],
                        myDataSet.Tables["tbl_products"].Columns["SupplierID"]
                    );

            foreach (DataRow pRow in myDataSet.Tables["tbl_suppliers"].Rows)
            {
                Console.WriteLine(pRow["CompanyName"]);
                foreach (DataRow cRow in pRow.GetChildRows(dr))
                {
                    Console.WriteLine("\t" + cRow["ProductName"]);
                }
            }

            Console.ReadKey();
        }
```

```
}
```

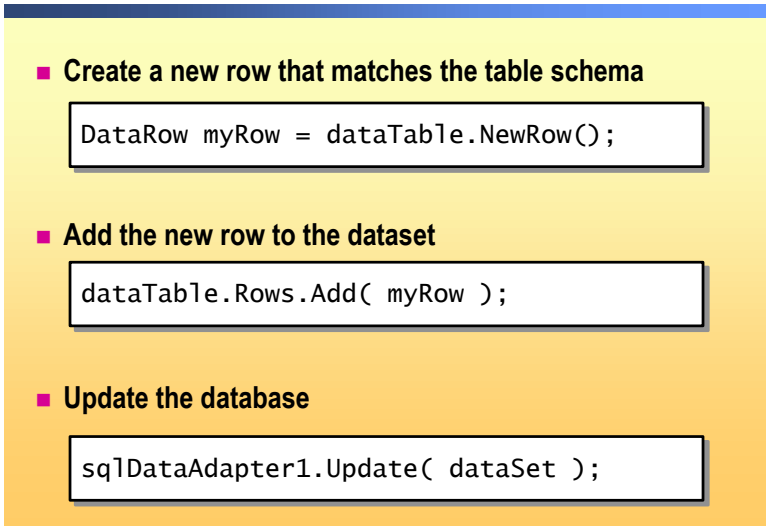
How to Update a Database in ADO.NET



Updating a database in ADO.NET involves three steps:

1. Update the data in the **dataset.**
2. Update the data in the **database**.
3. Notify the dataset that the database **accepted the changes.**

How to Create a Database Record



Example 1:

```
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = @"data source=localhost; Initial catalog=Northwind;
integrated security=SSPI";
            string commandString = @"SELECT * FROM Shippers";

            SqlDataAdapter dataAdapter = new SqlDataAdapter(commandString,
connectionString);

            SqlCommandBuilder scb = new SqlCommandBuilder(dataAdapter);

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill(myDataSet, "Shippers");

            DataTable sTable = myDataSet.Tables["Shippers"];

            // add data
            object[] o = { 0, "General", "555-1212" };
            sTable.Rows.Add(o);

            dataAdapter.Update(myDataSet, "Shippers");

            myDataSet.AcceptChanges();

        }
    }
```

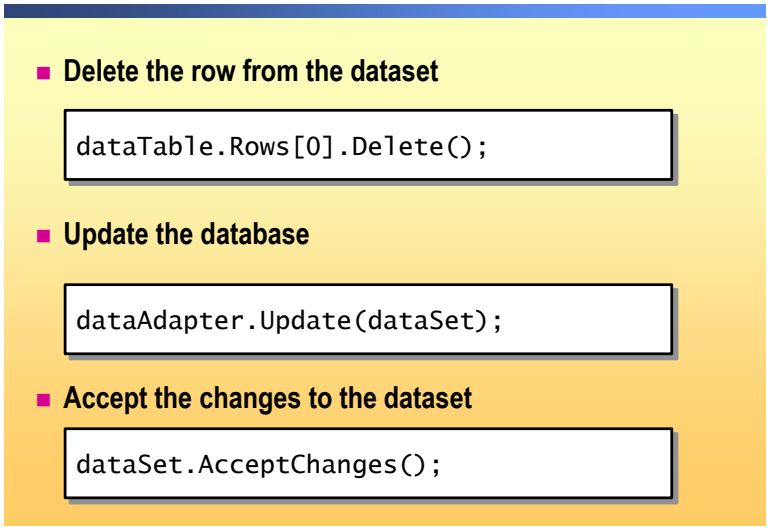
Modify a Database record

1. Make the desired **modifications** to the **rows** in the **DataSet**.
2. Generate a **new dataset that contains only the changed records** (rows).
3. Examine this dataset for errors, and then fix any errors that you can.
4. If you have fixed any changes, **merge the new dataset back into the original dataset.**
5. Call the **Update** method of the data adapter to merge the changes back into the database.
6. If the changes to the database were successful, accept them in the dataset by calling the **AcceptChanges** method. If they were unsuccessful, reject them by calling the **RejectChanges** method.

```
   class Program
    {
        static void Main(string[] args)
        {
            string connectionString = "data source=localhost;database=ERPDB;Integrated
Security=false; user id=sa; password=leads@123";
            string commandString = @"SELECT * FROM Suppliers";
            SqlDataAdapter dataAdapter = new SqlDataAdapter(commandString,
connectionString);
            SqlCommandBuilder scb = new SqlCommandBuilder(dataAdapter);

            DataSet myDataSet = new DataSet();
            dataAdapter.Fill(myDataSet, "Suppliers");

            DataTable sTable = myDataSet.Tables["Suppliers"];

            try
            {
                DataRow targetRow = sTable.Rows[1];

                targetRow.BeginEdit();
                targetRow["CompanyName"] = "Standard";
                targetRow.EndEdit();

                DataSet changedSet;
                changedSet = myDataSet.GetChanges(DataRowState.Modified);

                if (changedSet == null)
                    return;

                // check for errors
                bool _fixed = false;
                if (changedSet.HasErrors)
                {
                    // Fix errors setting fixed=true
                    // or else return;
                    return;
                }

                if (_fixed)
                {
```

```
                myDataSet.Merge(changedSet);
            }
            int n = dataAdapter.Update(myDataSet, "Suppliers");

            if (n > 0)
            {
                myDataSet.AcceptChanges();
            }

        }
        catch (Exception exp)
        {
            myDataSet.RejectChanges();
        }
        Console.ReadKey();
    }

}
```

How to Delete a Database Record




```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Address { get; set; }

}


public class CustomerDAL
{
```

```
        //string connectionString = "data
source=192.168.20.125;database=TestDB;Integrated Security=false; user id=sa;
password=leads@123";
        string connectionString =
ConfigurationManager.ConnectionStrings["SQLServerConnectionString"].ToString();
        private SqlConnection sqlConn = null;
        private SqlCommand cmd = null;
        string msg = string.Empty;

        public CustomerDAL()
        {
            sqlConn = new SqlConnection(connectionString);
            cmd = new SqlCommand();
            cmd.Connection = sqlConn;
        }

        public string InsertCustomer(Customer objCustomer)
        {
            string msg = String.Empty;

            int noOfRowEffected = 0;

            try
            {
                if (sqlConn.State == ConnectionState.Closed)
                    sqlConn.Open();
                cmd.CommandType = CommandType.Text;
                cmd.CommandText = "INSERT INTO Customers(Name , Age , Address)"
                                  + " VALUES('" + objCustomer.Name + "'," +
objCustomer.Age + ",'" + objCustomer.Address + "')";
                noOfRowEffected = cmd.ExecuteNonQuery();
                cmd.Parameters.Clear();
            }
            catch (Exception exp)
            {
                throw (exp);
            }
            finally
            {
                if (sqlConn.State == ConnectionState.Open)
                    sqlConn.Close();
            }

            if (noOfRowEffected > 0)
                return "Customer information saved successfully!";
            else
                return msg;
        }

        public DataTable GetAllCustomer()
        {
            DataTable tblCustomers = new DataTable();
            SqlDataReader rdr = null;

            cmd.CommandType = CommandType.Text;
            cmd.CommandText = "SELECT [Id] ,[Name],[Age] ,[Address] ,[IntroducerId] FROM
[dbo].[Customers]";
```

```
            try
            {
                if (sqlConn.State == ConnectionState.Closed)
                    sqlConn.Open();

                rdr = cmd.ExecuteReader();
                tblCustomers.Load(rdr);
            }
            catch (Exception exp)
            {
                throw (exp);
            }
            finally
            {
                if (sqlConn.State == ConnectionState.Open)
                    sqlConn.Close();
            }

            return tblCustomers;

        }

        public string UpdateCustomer(Customer objCustomer)
        {
            string msg = String.Empty;

            int noOfRowEffected = 0;

            try
            {
                if (sqlConn.State == ConnectionState.Closed)
                    sqlConn.Open();
                cmd.CommandType = CommandType.Text;
                cmd.CommandText = "UPDATE Customers SET Name = '" + objCustomer.Name +
                                    "',Age = " + objCustomer.Age + ", Address = '" +
objCustomer.Address
                                    + "' WHERE Id = " + objCustomer.Id;

                noOfRowEffected = cmd.ExecuteNonQuery();
                cmd.Parameters.Clear();
            }
            catch (Exception exp)
            {
                throw (exp);
            }
            finally
            {
                if (sqlConn.State == ConnectionState.Open)
                    sqlConn.Close();
            }

            if (noOfRowEffected > 0)
                return "Customer information updated successfully!";
            else
                return msg;
        }
```

```
        public string DeleteCustomer(int Id)
        {
            string msg = String.Empty;

            int noOfRowEffected = 0;

            try
            {
                if (sqlConn.State == ConnectionState.Closed)
                    sqlConn.Open();
                cmd.CommandType = CommandType.Text;
                cmd.CommandText = "DELETE FROM Customers"
                                    + " WHERE Id = " + Id;

                noOfRowEffected = cmd.ExecuteNonQuery();
                cmd.Parameters.Clear();
            }
            catch (Exception exp)
            {
                throw (exp);
            }
            finally
            {
                if (sqlConn.State == ConnectionState.Open)
                    sqlConn.Close();
            }

            if (noOfRowEffected > 0)
                return "Customer information deleted successfully!";
            else
                return msg;
        }
    }

    public class CustomerBLL
    {
        CustomerDAL objCustomerDAL;

        public CustomerBLL()
        {
            objCustomerDAL = new CustomerDAL();
        }

        public string AddCustomer(Customer objCustomer)
        {
            try
            {
                return objCustomerDAL.InsertCustomer(objCustomer);
            }
            catch (Exception exp)
            {
                throw (exp);
            }
        }

        public DataTable GetAllCustomer()
        {
            try
```

```
            {
                return objCustomerDAL.GetAllCustomer();
            }
            catch (Exception exp)
            {
                throw (exp);
            }
        }

        public string EditCustomer(Customer objCustomer)
        {
            try
            {
                return objCustomerDAL.UpdateCustomer(objCustomer);
            }
            catch (Exception exp)
            {
                throw (exp);
            }
        }

        public string RemoveCustomer(int Id)
        {
            try
            {
                return objCustomerDAL.DeleteCustomer(Id);
            }
            catch (Exception exp)
            {
                throw (exp);
            }
        }

    }


    class Program
    {
        public static void DisplayCustomerInfo(DataTable dataTable)
        {
            DataColumn column = dataTable.Columns[0];
            Console.WriteLine(dataTable.Columns[0].ColumnName + "      " +
dataTable.Columns[1].ColumnName +"      "
            + dataTable.Columns[2].ColumnName + "      " +
dataTable.Columns[3].ColumnName + "      ");

Console.WriteLine("_____________________________________________________________");

            foreach (DataRow row in dataTable.Rows)
            {
                Console.WriteLine(row[0].ToString() + " " + row[1].ToString() + " " +
row[2].ToString() + " " + row[3].ToString());
            }
        }

        static void Main(string[] args)
        {
            string msg = String.Empty;
```

```
            Customer objCustomer = new Customer();
            objCustomer.Name = "Jamil";
            objCustomer.Age = 28;
            objCustomer.Address = "Rangpur";

            CustomerBLL objCustomerBLL = new CustomerBLL();
            msg = objCustomerBLL.AddCustomer(objCustomer);
            Console.WriteLine(msg);

            DisplayCustomerInfo(objCustomerBLL.GetAllCustomer());

            objCustomer.Id = 2;
            objCustomer.Name = "Jamal Hossain";
            objCustomer.Age = 23;
            objCustomer.Address = "Chandpur";

            msg = objCustomerBLL.EditCustomer(objCustomer);
            Console.WriteLine(msg);

            DisplayCustomerInfo(objCustomerBLL.GetAllCustomer());

            msg = objCustomerBLL.RemoveCustomer(3);
            Console.WriteLine(msg);

            DisplayCustomerInfo(objCustomerBLL.GetAllCustomer());

            Console.ReadKey();

        }
    }
```

App.Config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
  <connectionStrings>
    <add name="SQLServerConnectionString" connectionString="data
source=192.168.20.125;database=TestDB;Integrated Security=false; user id=sa;
password=leads@123" providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

Indexers

- Enable user to create a class, struct, or interface that client applications can access just as an array
- Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array
- An indexer is a property that allows you to index an object in the same way as an array.
- The indexer notation not only simplifies the syntax for client applications; it also makes the class and its purpose more intuitive for other developers to understand

Syntax of Indexer

```
public int this[int index]    // Indexer declaration
{
    // get and set accessors
}
```

Example: Index an object

```
public abstract class Animal
{
    abstract public void Eat();
}

public class Lion : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Lion is eating.");
    }
}
public class Elephant : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Elephant is eating.");
    }
}

public class Antelope : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Antelope is eating.");
    }
}


public class Zoo
{
    private Animal[] theAnimals;

    public Animal this[int i]
    {
        set
        {
            theAnimals[i] = value;
        }
        get
        {
            return theAnimals[i];
        }

    }
```

```
    public Zoo()
    {
        // Our Zoo can hold 100 animals
        theAnimals = new Animal[100];
    }
}

class ZooKeeper
{
    static void Main(string[] args)
    {
        Zoo myZoo = new Zoo();
        myZoo[0] = new Elephant();
        myZoo[1] = new Lion();
        myZoo[2] = new Lion();
        myZoo[3] = new Antelope();

        Animal oneAnimal = myZoo[3];

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