Process and Thread

A process represents an application whereas a thread represents a module of the application. Process is heavyweight component whereas thread is lightweight. A thread can be termed as lightweight subprocess because it is executed inside a process.

# **C# Multithreading**

Multithreading in C# is a process in which multiple threads work simultaneously. It is a process to achieve multitasking. It saves time because multiple tasks are being executed at a time. To create multithreaded application in C#, we need to use **System.Threding** namespace.

## System.Threading Namespace

The System.Threading namespace contains classes and interfaces to provide the facility of multithreaded programming. It also provides classes to synchronize the thread resource. A list of commonly used classes are given below:

* Thread
* Mutex
* Timer
* Monitor
* Semaphore
* ThreadLocal
* ThreadPool
* Volatile etc.

# **C# Multithreading**

Multithreading in C# is a process in which multiple threads work simultaneously. It is a process to achieve multitasking. It saves time because multiple tasks are being executed at a time. To create multithreaded application in C#, we need to use **System.Threding** namespace.

## System.Threading Namespace

The System.Threading namespace contains classes and interfaces to provide the facility of multithreaded programming. It also provides classes to synchronize the thread resource. A list of commonly used classes are given below:

* Thread
* Mutex
* Timer
* Monitor
* Semaphore
* ThreadLocal
* ThreadPool
* Volatile etc.

## Unstarted State

When the instance of Thread class is created, it is in unstarted state by default.

## Runnable State

When start() method on the thread is called, it is in runnable or ready to run state.

## Running State

Only one thread within a process can be executed at a time. At the time of execution, thread is in running state.

## Not Runnable State

The thread is in not runnable state, if sleep() or wait() method is called on the thread, or input/output operation is blocked.

## Dead State

After completing the task, thread enters into dead or terminated state.

[**next →**](https://www.javatpoint.com/c-sharp-thread-synchronization)[**← prev**](https://www.javatpoint.com/c-sharp-thread-name)

# **C# Threading Example: ThreadPriority**

Let's see an example where we are changing the priority of the thread. The high priority thread can be executed first. But it is not guaranteed because thread is highly system dependent. It increases the chance of the high priority thread to execute before low priority thread.

1. **using** System;
2. **using** System.Threading;
3. **public** **class** MyThread
4. {
5. **public** **void** Thread1()
6. {
7. Thread t = Thread.CurrentThread;
8. Console.WriteLine(t.Name+" is running");
9. }
10. }
11. **public** **class** ThreadExample
12. {
13. **public** **static** **void** Main()
14. {
15. MyThread mt = **new** MyThread();
16. Thread t1 = **new** Thread(**new** ThreadStart(mt.Thread1));
17. Thread t2 = **new** Thread(**new** ThreadStart(mt.Thread1));
18. Thread t3 = **new** Thread(**new** ThreadStart(mt.Thread1));
19. t1.Name = "Player1";
20. t2.Name = "Player2";
21. t3.Name = "Player3";
22. t3.Priority = ThreadPriority.Highest;
23. t2.Priority = ThreadPriority.Normal;
24. t1.Priority = ThreadPriority.Lowest;
26. t1.Start();
27. t2.Start();
28. t3.Start();
29. }
30. }

Output:

The output is unpredictable because threads are highly system dependent. It may follow any algorithm preemptive or non-preemptive.

# **C# Inheritance**

In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.

In C#, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base** class. The derived class is the specialized class for the base class.

### **Advantage of C# Inheritance**

**Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

## C# Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

1. **using** System;
2. **public** **class** Employee
3. {
4. **public** **float** salary = 40000;
5. }
6. **public** **class** Programmer: Employee
7. {
8. **public** **float** bonus = 10000;
9. }
10. **class** TestInheritance{
11. **public** **static** **void** Main(**string**[] args)
12. {
13. Programmer p1 = **new** Programmer();
15. Console.WriteLine("Salary: " + p1.salary);
16. Console.WriteLine("Bonus: " + p1.bonus);
18. }
19. }

Output:

Salary: 40000

Bonus: 10000

In the above example, Employee is the **base** class and Programmer is the **derived** class.

# **C# Polymorphism**

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation and polymorphism.

There are two types of polymorphism in C#: compile time polymorphism and runtime polymorphism. Compile time polymorphism is achieved by method overloading and operator overloading in C#. It is also known as static binding or early binding. Runtime polymorphism in achieved by method overriding which is also known as dynamic binding or late binding.

### **C# Runtime Polymorphism Example**

Let's see a simple example of runtime polymorphism in C#.

Volume 0%

1. **using** System;
2. **public** **class** Animal{
3. **public** **virtual** **void** eat(){
4. Console.WriteLine("eating...");
5. }
6. }
7. **public** **class** Dog: Animal
8. {
9. **public** **override** **void** eat()
10. {
11. Console.WriteLine("eating bread...");
12. }
14. }
15. **public** **class** TestPolymorphism
16. {
17. **public** **static** **void** Main()
18. {
19. Animal a= **new** Dog();
20. a.eat();
21. }
22. }

Output:

eating bread...

[**next →**](https://www.javatpoint.com/c-sharp-method-overriding)[**← prev**](https://www.javatpoint.com/c-sharp-aggregation)

# **C# Member Overloading**

If we create two or more members having same name but different in number or type of parameter, it is known as member overloading. In C#, we can overload:

* methods,
* constructors, and
* indexed properties

It is because these members have parameters only.

## C# Method Overloading

Having two or more methods with same name but different in parameters, is known as method overloading in C#.

The **advantage** of method overloading is that it increases the readability of the program because you don't need to use different names for same action.

You can perform method overloading in C# by two ways:

1. By changing number of arguments
2. By changing data type of the arguments

### **C# Method Overloading Example: By changing no. of arguments**

Let's see the simple example of method overloading where we are changing number of arguments of add() method.

1. **using** System;
2. **public** **class** Cal{
3. **public** **static** **int** add(**int** a,**int** b){
4. **return** a + b;
5. }
6. **public** **static** **int** add(**int** a, **int** b, **int** c)
7. {
8. **return** a + b + c;
9. }
10. }
11. **public** **class** TestMemberOverloading
12. {
13. **public** **static** **void** Main()
14. {
15. Console.WriteLine(Cal.add(12, 23));
16. Console.WriteLine(Cal.add(12, 23, 25));
17. }
18. }

Output:

35

60

### **Member Overloading Example: By changing data type of arguments**

Let's see the another example of method overloading where we are changing data type of arguments.

1. **using** System;
2. **public** **class** Cal{
3. **public** **static** **int** add(**int** a, **int** b){
4. **return** a + b;
5. }
6. **public** **static** **float** add(**float** a, **float** b)
7. {
8. **return** a + b;
9. }
10. }
11. **public** **class** TestMemberOverloading
12. {
13. **public** **static** **void** Main()
14. {
15. Console.WriteLine(Cal.add(12, 23));
16. Console.WriteLine(Cal.add(12.4f,21.3f));
17. }
18. }

Output:

35

33.7

[**next →**](https://www.javatpoint.com/c-sharp-base)[**← prev**](https://www.javatpoint.com/c-sharp-member-overloading)

# **C# Method Overriding**

If derived class defines same method as defined in its base class, it is known as method overriding in C#. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the method which is already provided by its base class.

To perform method overriding in C#, you need to use **virtual** keyword with base class method and **override** keyword with derived class method.

## C# Method Overriding Example

Let's see a simple example of method overriding in C#. In this example, we are overriding the eat() method by the help of override keyword.

1. **using** System;
2. **public** **class** Animal{
3. **public** **virtual** **void** eat(){
4. Console.WriteLine("Eating...");
5. }
6. }
7. **public** **class** Dog: Animal
8. {
9. **public** **override** **void** eat()
10. {
11. Console.WriteLine("Eating bread...");
12. }
13. }
14. **public** **class** TestOverriding
15. {
16. **public** **static** **void** Main()
17. {
18. Dog d = **new** Dog();
19. d.eat();
20. }
21. }

Output:

Eating bread...

# **C# Enum**

Enum in C# is also known as enumeration. It is used to store a set of named constants such as season, days, month, size etc. The enum constants are also known as enumerators. Enum in C# can be declared within or outside class and structs.

Enum constants has default values which starts from 0 and incremented to one by one. But we can change the default value.

### **Points to remember**

* enum has fixed set of constants
* enum improves type safety
* enum can be traversed

### **C# Enum Example**

Let's see a simple example of C# enum.

1. **using** System;
2. **public** **class** EnumExample
3. {
4. **public** **enum** Season { WINTER, SPRING, SUMMER, FALL }
6. **public** **static** **void** Main()
7. {
8. **int** x = (**int**)Season.WINTER;
9. **int** y = (**int**)Season.SUMMER;
10. Console.WriteLine("WINTER = {0}", x);
11. Console.WriteLine("SUMMER = {0}", y);
12. }
13. }

Output:

WINTER = 0

SUMMER = 2

# **C# Encapsulation**

Encapsulation is the concept of wrapping data into a single unit. It collects data members and member functions into a single unit called class. The purpose of encapsulation is to prevent alteration of data from outside. This data can only be accessed by getter functions of the class.

A fully encapsulated class has getter and setter functions that are used to read and write data. This class does not allow data access directly.

Here, we are creating an example in which we have a class that encapsulates properties and provides getter and setter functions.

### **Example**

1. **namespace** AccessSpecifiers
2. {
3. **class** Student
4. {
5. // Creating setter and getter for each property
6. **public** **string** ID { **get**; **set**; }
7. **public** **string** Name { **get**; **set**; }
8. **public** **string** Email { **get**; **set**; }
9. }
10. }
11. **using** System;
12. **namespace** AccessSpecifiers
13. {
14. **class** Program
15. {
16. **static** **void** Main(**string**[] args)
17. {
18. Student student = **new** Student();
19. // Setting values to the properties
20. student.ID = "101";
21. student.Name = "Mohan Ram";
22. student.Email = "mohan@example.com";
23. // getting values
24. Console.WriteLine("ID = "+student.ID);
25. Console.WriteLine("Name = "+student.Name);
26. Console.WriteLine("Email = "+student.Email);
27. }
28. }
29. }

**Output:**

ID = 101

Name = Mohan Ram

Email = mohan@example.com

# stored Procedures in C#

ADO.NET consists of a set of Classes that interact to provide the required functionality. A stored procedure is a group of Transact-SQL statements compiled into a single execution plan. A sample Stored Procedure is given below :



The above code create a procedure named as 'SPPUBLISHER' and it execute SQL statement that select all publisher name from publishers table from the PUB database. Coding business logic into a single stored procedure also offers a single point of control for ensuring that business rules are correctly enforced.

The command Object in ADO.NET provides a number of Execute methods that can be used to perform the SQL queries in a variety of fashions. To call a stored procedure from C# , set the CommandType of the Command object to StoredProcedure.



From the following source code you can see how to call a stored procedure from a C# application.

# csssss# Stored Procedure with Parameter

The .NET Data Providers consist of a number of classes used to connect to a data source, execute commands, and return recordsets. The Command Object in ADO.NET provides a number of Execute methods that can be used to perform the SQL queries in a variety of fashions.

A stored procedure is a pre-compiled executable object that contains one or more SQL statements. In many cases stored procedures accept input parameters and return multiple values . Parameter values can be supplied if a stored procedure is written to accept them. A sample stored procedure with accepting input parameter is given below :



The above stored procedure is accepting a country name (@COUNTRY VARCHAR(20)) as parameter and return all the publishers from the input country. Once the CommandType is set to StoredProcedure, you can use the Parameters collection to define parameters.



The above code passing country parameter to the stored procedure from C# application.