**Classes and Objects**

**Lesson Plan**

Today we will start Object Oriented Programming (OOP) in C#. We will start with learning classes, objects, and their basics. Then we will move to constructors, access modifiers, properties, method overloading and static methods.

**Concept of a Class**

A class is simply an abstract model used to define a new data types. A class may contain any combination of encapsulated data (fields or member variables), operations that can be performed on data (methods) and accessors to data (properties). For example, there is a class String in the System namespace of .Net Framework Class Library (FCL). This class contains an array of characters (data) and provide different operations (methods) that can be applied to its data like ToLowerCase(), Trim(), Substring(), etc. It also has some properties like Length (used to find the length of the string).

A class in C# is declared using the keyword class and its members are enclosed in parenthesis

class MyClass

{

// fields, operations and properties go here

}

where MyClass is the name of class or new data type that we are defining here.

**Objects**

As mentioned above, a class is an abstract model. An object is the concrete realization or instance built on the model specified by the class. An object is created in the memory using the keyword ’new’ and is referenced by an identifier called a "reference".

MyClass myObjectReference = new MyClass();

In the line above, we made an object of type MyClass which is referenced by an identifier myObjectReference.

The difference between classes and implicit data types is that objects are reference types (passed by reference) while implicit data types are value type (passed by making a copy). Also, objects are created at the heap while implicit data types are stored on stack.



**Fields**

Fields are the data contained in the class. Fields may be implicit data types, objects of some other class, enumerations, structs or delegates. In the example below, we define a class named Student containing a student’s name, age, marks in maths, marks in English, marks in science, total marks, obtained marks and a percentage.

class Student

{

// fields contained in Student class

string name;

int age;

int marksInMaths;

int marksInEnglish;

int marksInScience;

int totalMarks = 300; // initialization

int obtainedMarks;

double percentage;

}

You can also initialize the fields with the initial values as we did in totalMarks in the example above. If you don’t initialize the members of the class, they will be initialized with their default values.

Default values for different data types are shown below:

|  |  |
| --- | --- |
| **Data Type**  | Default Value |
| int  | 0 |
| long  | 0 |
| float  | 0.0 |
| double  | 0.0 |
| bool  | False |
| char  | ’\0’ (null character) |
| string  | "" (empty string) |
| Objects  | Null |

**Methods**

Methods are the operations performed on the data. A method may take some input values through its parameters and may return a value of a particular data type. The signature of the method takes the form

<return type> <name of method>(<data type> <identifier>, <data type> <identifier>,...)

{

// body of the method

}

For example,

int FindSum(int num1, int num2)

{

int sum = num1 + num2;

return sum;

}

Here, we defined a method named FindSum which takes two parameters of int type (num1 and num2) and returns a value of type int using the keyword return. If a method does not return anything, its return type would be void. A method can also optionally take no parameter (a parameterless method)

void ShowCurrentTime()

{

Console.WriteLine("The current time is: " + DateTime.Now);

}

The above method takes no parameter and returns nothing. It only prints the Current Date and Time on the console using the DateTime Class in the System namespace.

**Instantiating the class**

In C# a class is instantiated (making its objects) using the new keyword.

Student theStudent = new Student();

You can also declare the reference and assign an object to it in different steps. The following two lines are equivalent to the above line

Student theStudent;

theStudent = new Student();

Note that it is very similar to using implicit data types except for the object is created with the new operator while implicit data types are created using literals

int i;

i = 4;

Another important thing to understand is the difference between reference and object. The line

Student theStudent;

only declares the reference theStudent of type Student which at this point does not contain any object (and points to the default null value) so if you try to access the members of class (Student) through it, it will throw a compile time error ’Use of unassigned variable theStudent’. When we write

theStudent = new Student();

then a new object of type Student is created at the heap and its reference (or handle) is given to theStudent. Only now is it legal to access the members of the class through it.

**Accessing the members of a class**

The members of a class (fields, methods and properties) are accessed using dot ’.’ operator against the reference of the object like this:

Student theStudent = new Student();

theStudent.marksOfMaths = 93;

theStudent.CalculateTotal();

Console.WriteLine(theStudent.obtainedMarks);

Let’s now make our Student class with some related fields, methods and then instantiate it in the Main() method.

using System;

namespace CSharpSchool

{

// Defining a class to store and manipulate students information

class Student

{

// fields

string name;

int age;

int marksOfMaths;

int marksOfEnglish;

int marksOfScience;

int totalMarks = 300;

int obtainedMarks;

double percentage;

// methods

void CalculateTotalMarks()

{

obtainedMarks = marksOfMaths + marksOfEnglish + marksOfScience;

}

void CalculatePercentage()

{

percentage = (double) obtainedMarks / totalMarks \* 100;

}

double GetPercentage()

{

return percentage;

}

// Main method or entry point of program

static void Main()

{

// creating new instance of Student

Student st1 = new Student();

// setting the values of fields

st1.name = "Einstein";

st1.age = 20;

st1.marksOfEnglish = 80;

st1.marksOfMaths = 99;

st1.marksOfScience = 96;

// calling functions

st1.CalculateTotalMarks();

st1.CalculatePercentage();

double st1Percentage = st1.GetPercentage();

// calling and retrieving value

// returned by the function

Student st2 = new Student();

st2.name = "Newton";

st2.age = 23;

st2.marksOfEnglish = 77;

st2.marksOfMaths = 100;

st2.marksOfScience = 99;

st2.CalculateTotalMarks();

st2.CalculatePercentage();

double st2Percentage = st2.GetPercentage();

Console.WriteLine("{0} of {1} years age got {2}% marks", st1.name, st1.age, st1.percentage);

Console.WriteLine("{0} of {1} years age got {2}% marks", st2.name, st2.age, st2.percentage);

}

}

}

Here, we started by creating an object of the Student class (st1), we then assigned name, age and marks of the student. Later, we called methods to calculate totalMarks and percentage, then we retrieved and stored the percentage in a variable and finally printed these on a console window.

We repeated the same steps again to create another object of type Student, set and printed its attributes. Hence in this way, you can create as many object of the Student class as you want. When you compile and run this program it will display:



**Access Modifiers or Accessibility Levels**

In our Student class, everyone has access to each of the fields and methods. So if one wants, he/she can change the totalMarks from 300 to say 200, resulting in the percentages getting beyond 100%, which in most cases we like to restrict. C# provides access modifiers or accessibility levels just for this purpose, i.e., restricting access to a particular member. There are 5 access modifiers that can be applied to any member of the class. We are listing these along with short description in the order of decreasing restriction

|  |  |
| --- | --- |
| Access Modifier | Description |
| private | private members can only be accessed within the class that contains them |
| protected internal | This type of member can be accessed from the current project or from the types inherited from their containing type |
| internal | Can only be accessed from the current project |
| protected | Can be accessed from a containing class and types inherited from the containing class |
| public | public members are not restricted to anyone. Anyone who can see them can also access them. |

In Object Oriented Programming (OOP) it is always advised and recommended to mark all your fields as private and allow the user of your class to access only certain methods by making them public. For example, we may change our student class by marking all the fields private and the three methods in the class public.

class Student

{

// fields

private string name;

private int age;

private int marksOfMaths;

private int marksOfEnglish;

private int marksOfScience;

private int totalMarks = 300;

private int obtainedMarks;

private double percentage;

// methods

public void CalculateTotalMarks()

{

obtainedMarks = marksOfMaths + marksOfEnglish + marksOfScience;

}

public void CalculatePercentage()

{

percentage = (double) obtainedMarks / totalMarks \* 100;

}

public double GetPercentage()

{

return percentage;

}

}

If you don’t mark any member of class with an access modifier, it will be treated as a private member; this means the default access modifier for the members of a class is private.

You can also apply access modifiers to other types in C# such as the class, interface, struct, enum, delegate and event. For top-level types (types not bound by any other type except namespace) like class, interface, struct and enum you can only use public and internal access modifiers with the same meaning as described above. In fact other access modifiers don’t make sense to these types. Finally you can not apply access modifiers to namespaces.

**Properties**

You must be wondering if we declare all the fields in our class as private, how can we assign values to them through their reference as we did in the Student class before? The answer is through Properties. C# is the first language to provide the support of defining properties in the language core.

In traditional languages like Java and C++, for accessing the private fields of a class, public methods called getters (to retrieve the value) and setters (to assign the value) were defined like if we have a private field name

private string name;

then, the getters and setters would be like

// getter to name field

public string GetName()

{

return name;

}

// setter to name field

public void SetName(string theName)

{

name = theName;

}

Using these we could restrict the access to a particular member. For example we can opt to only define the getter for the totalMarks field to make it read only.

private int totalMarks;

public int GetTotalMarks()

{

return totalMarks;

}

Hence outside the class, one can only read the value of totalMarks and cannot modify it. You can also decide to check some condition before assigning a value to your field

private int marksOfMaths;

public void SetMarksOfMaths(int marks)

{

if(marks >= 0 && marks <=100)

{

marksOfMaths = marks;

}

else

{

marksOfMaths = 0;

// or throw some exception informing user marks out of range

}

}

This procedure gives you a lot of control over how fields of your classes should be accessed and dealt in a program.

But, the problem is this you need to define two methods and have to prefix the name of your fields with Get or Set.

C# provides the built in support for these getters and setters in the form of properties. Properties are context sensitive constructs used to read, write or compute private fields of class and to achieve control over how the fields can be accessed.

**Using Properties**

The general Syntax for Properties is

<access modifier> <data type> <name of property>

{

get

{

// some optional statements

return <some private field>;

}

set

{

// some optional statements;

<some private field> = value;

}

}

Didn’t understand it? No problem. Let’s clarify it with an example: we have a private field name

private string name;

We decide to define a property for this providing both getters and setters. We will simply write

public string Name

{

get

{

return name;

}

set

{

name = value;

}

}

We defined a property called ’Name’ and provided both a getter and a setter in the form of get { } and set { } blocks.

Note that we called our property ’Name’ which is accessing the private field ’name’. It is becoming convention to name the property the same as the corresponding field but with first letter in uppercase (for name->Name, for percentage->Percentage). As properties are accessors to certain fields, they are mostly marked as public while the corresponding field is (and should be) mostly private. Finally note in the set { } block, we wrote

name = value;

Here, value is a keyword and contains the value passed when a property is called. In our program we will use our property as

Student theStudent = new Student();

theStudent.Name = "Faraz";

string myName = theString.Name;

theStudent.name = "Someone not Faraz"; // error

While defining properties, we said properties are context sensitive. When we write

theStudent.Name = "Faraz";

The compiler sees that the property Name is on the left hand side of assignment operator, so it will call the set { } block of the properties passing "Faraz" as a value (which is a keyword). In the next line when we write

string myName = theString.Name;

the compiler now sees that the property Name is on the right hand side of the assignment operator, hence it will call the get { } block of property Name which will return the contents of the private field name ("Faraz" in this case, as we assigned in line 2) which will be stored in the local string variable name. Hence, when compiler finds the use of a property, it checks in which context it is called and takes appropriate action with respect to the context.

The last line

theStudent.name = "Someone not Faraz"; // error

will generate a compile time error (if called outside the Student class) as the name field is declared private in the declaration of class.

You can give the definition of either of get { } or set { } block. If you miss one of these, and user tries to call it, he/she will get compile time error. For example the Length property in String class is read only; that is, the implementers have only given the definition of get { } block. You can write statements in the get { }, set { } blocks

as you do in methods.

private int marksOfMaths;

public int MarksOfMaths

{

set

{

if(value >= 0 && value<=100)

{

marksOfMaths = value;

}

else

{

marksOfMaths = 0;

// or throw some exception informing user marks out of range

}

}

}

**Precautions when using properties**

• Properties don’t have argument lists; set, get and value are keywords in C#

• The data type of value is the same as the type of property you declared when declaring the property

• ALWAYS use proper curly brackets { } and proper indentation while using properties.

• DON’T try to write the set { } or get { } block in a single line

• UNLESS your property only assigns and retrieve values from the private fields like

get { return name; }

set { name = value; }

Each object has a reference this which points to itself. Suppose in some method call, our object needs to pass itself, what would we do? Suppose in our class Student, we have a method Store() that stores the information of Student on the disk. In this method, we called another method Save() of FileSystem class which takes the object to store as its parameter.

class Student

{

string name = "Some Student";

int age;

public void Store()

{

FileSystem fs = new FileSystem();

fs.save(this);

}

}

We passed this as a parameter to the method Save() which points to the object itself.

class Test

{

public static void Main()

{

Student theStudent = new Student();

theStudent.Store();

}

}

Here, when Store() is called, the reference theStudent will be passed as a parameter to the Save() method in Store().

Conventionally, the parameters to constructors and other methods are named the same as the name of the fields they refer to and are distinguished only by using this reference.

class Student

{

private string name;

private int age;

public Student(string name, int age)

{

this.name = name;

this.age = age;

}

}

Here in the constructor when we use name or age, we actually get the variables passed in the method which overshadow the instance members (fields) with same name. Hence, to use our fields, we had to use this to distinguish our instance members (fields) with the members passed through the parameters.

This is an extremely useful, widely and commonly used construct. I recommend you practice with "this" for some time until you feel comfortable with it.

**Static Members of the class**

All the members of the classes that we have seen up till now are instance members, meaning they belong to the object being created. For example, if you have an instance field name in your Person class then each object of our Person class will have a separate field name of its own. There is another class of members which are called static.

Static members belong to the whole class rather than to individual object. For example, if you have a static phoneNumber field in your Student class, then there will be the single instance of this field and all the objects of this class will share this single field. Changes made by one object to phoneNumber will be realized by the other object. Static members are defined using keyword static

class Student

{

public static int phoneNumber;

public int rollNumber;

}

Static members are accessed with the name of class rather than reference to objects. Let’s make our Test class containing Main method

class Test

{

public static void Main()

{

Student st1 = new Student();

Student st2 = new Student();

st1.rollNumber = 3;

st2.rollNumber = 5;

Student.phoneNumber = 4929067;

}

}

Here you can see that the phoneNumber is accessed without any reference to the object but with the name of the class it belongs. Static methods are very useful while programming. In fact, the WriteLine() and ReadLine() methods that we are using from the start are static methods of Console class. That is the reason why we used to call them with reference to their class rather than making an object of the Console class. I hope now you are able to understand the syntax of the Main method in C# in full. It is declared static as CLR calls it without making any instance of our class. Static variables are useful when you want to cache data that should be available to all objects of the class. You can use static fields, methods, properties and even constructors which will be called before any instance of the class is created. Static constructor are declared like

static Student()

{

name="unknown";

}

As static methods may be called without any reference to object, you cannot use instance members inside static methods or properties, while you may call a static member from a non-static context. The reason for being able to call static members from non-static context is that static members belong to the class and are present irrespective of the existence of even a single object. The definition of MyMethod() in following code will not compile

class Student

{

public static int phoneNumber;

public int rollNumber;

public void DoWork()

{

// legal, static method called in non-static context

MyMethod();

}

public static void MyMethod()

{

// legal, static field used in static context

phoneNumber++;

// illegal, non-static field used in static context

rollNumber++;

// illegal, non-static method used in static context

DoWork();

}

}

Some precautionary points in the end

• Don’t put too many static methods in your class as it is against the object oriented design principles and makes your class less extensible.

• Don’t try to make a class with only the static methods and properties unless you have very good reason for doing this.

• You tend to lose a number of object oriented advantages while using static methods, as static methods can’t be overridden which means it cannot be used polymorphically, something widely used in the Object Oriented Paradigm of programming.

**Some More about Methods**

We mentioned earlier that there are two kinds of ’types’ in C#: Value types and Reference types. Value types, such as implicit data types, are passed to methods by value. Reference types, like objects and arrays, are passed by reference.

**Constructors**

Constructors are a special kind of method. A Constructor has the following properties:

• It has the same name as its containing class

• It has no return type

• It is automatically called when a new instance or object of a class is created, hence why it’s called a constructor.

• The constructor contains initialization code for each object, like assigning default values to the fields.

Let us see some examples.

using System;

class Person

{

// field

private string name;

// constructor

public Person()

{

name = "unknown";

Console.WriteLine("Constructor called...");

}

// property

public string Name

{

get { return name; }

set { name = value; }

}

}

In the Person class above, we have a private field name, a public constructor which initializes the name field with string "unknown" and prints that it has been called, then we have a public property to read/write the private field name. Let’s make another class Test which contains the Main() method and which uses the Person class

class Test

{

public static void Main()

{

Person thePerson = new Person();

Console.WriteLine("The name of person in object thePerson is " + thePerson.Name);

thePerson.Name = "Faraz";

Console.WriteLine("The name of person in object thePerson is " + thePerson.Name);

}

}

In our Test class, we made an object of the Person class and printed the name of person. We then changed the value of Name and printed the Name again. The result of the program is:



Note that the constructor is called just as we created a new instance of Person class and initialized the field name with string "unknown". In fact, when we create a new object, we actually call the constructor of the class:

Person thePerson = new Person();

That is why constructor is usually made public. If you make your constructor private, no one would be able to make an object of your class outside of it (though a method in the class of course could). That is, if the Person class is defined as:

class Person

{

private Person()

{

}

}

then it would cause an error to write:

class Test

{

public static void Main()

{

Person thePerson = new Person(); // error

}

}

The constructors shown so far have been parameter-less, i.e. they do not take any parameters. We can define constructors which take some parameters.

class Person

{

private string name;

public Person(string theName)

{

name = theName;

Console.WriteLine("Constructor called...");

}

}

Now, the object of class Person can only be created by passing a string into the constructor.

Person thePerson = new Person("Faraz");

If you don’t define any constructor for your class, the compiler will generate an empty parameter-less constructor for you. That is why we were able to make our Student object even we did not specify any constructor for the Student class.

**Finalize() Method of Object class**

Each class in C# is automatically (implicitly) inherited from the Object class which contains a method Finalize().

This method is guaranteed to be called when your object is garbage collected (removed from memory). You can override this method and put here code for freeing resources that you reserved when using the object. For example,

protected override void Finalize()

{

try

{

Console.WriteLine("Destructing object...");

// put some code here

}

finally

{

base.Finalize();

}

}

**Author’s Note:** I am not going to explain this code for now. If it looks alien to you, read it again when we would have covered inheritance, polymorphism and exceptions. We will explain Garbage Collection in coming issues.

**Destructors**

Destructors are just the opposite of constructors. These are methods with the following properties:

* It has the same name as the containing class but prefixes it with the ~ (tilde) sign.
* It is called automatically when the object is about to be destructed (when garbage collector is about to destroy your object).
* It has no return type.

We declare the destructor as

class Person

{

// constructor

public Person()

{

}

// destructor

~Person()

{

// put resource freeing code here.

}

}

As a matter of fact, the C# compiler internally converts the destructor to the Finalize() method, we just saw above.

Destructors are not used very much in common C# programming practice (that is why Java dropped the idea of destructors). In the days of C++, programmers had to manage memory allocation and de-allocation. Destructors were used there to free the memory allocated by the object dynamically. Hence, you probably won’t encounter destructors or Finalize() methods that often.

**Method and Constructor Overloading**

It is possible to have more than one method with the same name and return type but with a different number and type of arguments (parameters). This is called method overloading. For example it is perfectly legal to write:

class Checker

{

// 1st overloaded form

public bool isDefaultValue(bool val)

{

if(val == false)

return true;

else

return false;

}

// 2nd overloaded form

public bool isDefaultValue(int val)

{

if(val == 0)

return true;

else

return false;

}

// 3rd overloaded form

public bool isDefaultValue(int intVal, bool booleanVal)

{

if(intVal == 0 && booleanVal == false)

return true;

else

return false;

}

}

In the checker class above we defined three methods with the name isDefaultValue(). The return type of all these is bool but all differ from each other in parameter list. The first two differ in the data type of the parameters while the third one differs in the number of parameters. When isDefaultValue() is called, the compiler will decide (on the basis of the types and number of parameters being passed) which one of these three to actually call. For example, in our Main() method:

Checker check = new Checker();

Console.WriteLine(check.isDefaultValue(5)); // calls the first one

Console.WriteLine(check.isDefaultValue(false)); // calls the second one

Console.WriteLine(check.isDefaultValue(0, true)); // calls the third one

Remember that methods are overloaded depending on the parameter list and not on the return type. The WriteLine() method of Console class in the System namespace has 19 different overloaded forms! See the .Net Framework Documentation or MSDN for all of these.

**Overloading Constructors**

Since constructors are a special type of method, we can overload constructors similarly.

class Person

{

private string name;

public Person()

{

name = "uknown";

}

public Person(string theName)

{

name = theName;

}

}

Now, if we create an object like

Person thePerson = new Person();

the first constructor will be called initializing name with "unknown". If we create an object like

Person thePerson = new Person("Faraz");

the second constructor will be called initializing name with "Faraz". As you can see, overloading methods and constructors gives your program a lot of flexibility and reduces a lot of complexity that would otherwise be produced if we had to use different name for these methods (Just consider what would happen to implementers of

WriteLine(), who would have had to come up with 19 names!)

**Value types (out & ref Keywords)**

When we pass a variable of an implicit data type to a method, conceptually the runtime generates a copy and passes that copy to the method. It is actually a copy of the variable that is available inside the method. Hence if you modify a value type variable (passed as a parameter) in a method, the actual value of the variable would not be changed outside the method. Let us have in our test class a Main() method and a DoWork() method:

class Test

{

public static void Main()

{

int a = 3;

DoWork(a);

Console.WriteLine("The value of a is " + a);

}

public static void DoWork(int i)

{

i++;

}

}

The program will result in



The value of a is 3

Because a copy of the variable a is passed to the DoWork() method and not the variable a. Also, note that i is the local variable in DoWork() and a is a local variable in Main(). Hence, they can be accessed within their containing methods only. In fact, we may define int a; in different methods and each will have its own variable a and none would have correspondence with any other implicitly.

C# provides a keyword, ref, which means that the value type will be passed by reference instead of the default by value behavior. Hence, changes done inside the method would be reflected back after the method has been called and terminated. Both the method signature and method calling should be declared as ref in order to override the by value characteristic to by ref.

class Test

{

public static void Main()

{

int a = 3; // must be initialized

DoWork(ref a); // note ref

Console.WriteLine("The value of a is " + a);

}

public static void DoWork(ref int i) // note ref

{

i++;

}

}

The program will give the following result:



In the case of the ref keyword, the variable must be initialized before passing it to the method by reference. C# also provides the out keyword. This is used for passing a variable for output purposes. This will again be passed by reference. However, when using the out keyword, it is not necessary to initialize the variable.

class Test

{

public static void Main()

{

int a; // may be left un-initialized

DoWork(out a); // note out

Console.WriteLine("The value of a is " + a);

}

public static void DoWork(out int i) // note out

{

i=4;

}

}

The program will give the result



**Reference types**

Objects are implicitly passed by reference. This means that only a copy of the reference is passed to the methods during method invocation. Hence, if we initialize an array (which is an object in C#) and pass it to some method where the array gets changed, then this changed effect would be visible after the method has been terminated in the calling method.

class Test

{

public static void Main()

{

int [] nums = { 2, 4, 8 } ;

DoWork(nums);

int count =0;

foreach(int num in nums)

Console.WriteLine("The value of a[{0}] is {1}", count++, num);

}

public static void DoWork(int [] numbers)

{

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

numbers[i]++;

}

}

The program will result in



Here, we initialized an int type array (nums) with some values. We passed this array to the DoWork() method, which incremented (modified) the contents of the array. Finally, we printed the elements of array. As the output suggests, the DoWork() method did change the elements in array (num) and worked on actual array and not on its copy (as is the case in value types).

**Some more about references and objects**

A reference is just a pointer or handle to the object in memory. It is possible to create an object without giving its handle to any reference:

new Student();

The above line is a valid statement. It will create an object of Student class without any reference pointing to it. An object is actually eligible to be garbage collected when there is no reference to point it. So, in the above case, the new Student object will instantly be eligible to be garbage collected after its creation. Experienced programmers often call methods on these unreferenced objects like

int pc = ( new Student(87, 94, 79) ).CalculatePercentage();

In the above line, a new object of class Studrent is created and the CalculatePercentage() method is called on it.

This newly created, unreferenced object will be eligible to be garbage collected just after the method CalculatePercentage() completes its execution. I personally won’t encourage you to write such statements. The above line is similar to

Student theStduent = new Student(87, 94, 79);

int pc = theStudent.CalculatePercentage();

theStudent = null;

We assigned null to theStudent so the object above will be destroyed after method call terminates as in the case of previous example. When you write:

Student student1 = new Student("Faraz");

a new object of type Student is created at the heap and its handle is given to the reference student1. Let us make another object and give its handle to the reference student2.

Student student2 = new Student("Newton");

Now, if we write

Student student3 = student2;

The new reference student3 will also start pointing the student (Newton) already pointed by student2. Hence both student2 and student3 will be pointing to same student and both can make changes to same object.



Now if we write,

student1 = student3;

it will also start pointing to the second object (Newton), leaving the first student (Faraz) unreferenced. It means the first student is now eligible to be garbage collected and can be removed from memory anytime.



If you want a reference to reference nothing, you can set it to null, which is a keyword in C#.

student1 = null;