

Boxing & unboxing (Value to reference type & vice versa):

In C#, **boxing** converts a value type (like `int`, `double`, `char`) into an object type (`object` or `System.Object`).

```
int num = 10;  
object obj = num;
```

Unboxing:

Unboxing is the process of extracting the value type from an object. It requires explicit type casting.

```
object obj = 10;  
int num = (int)obj;
```

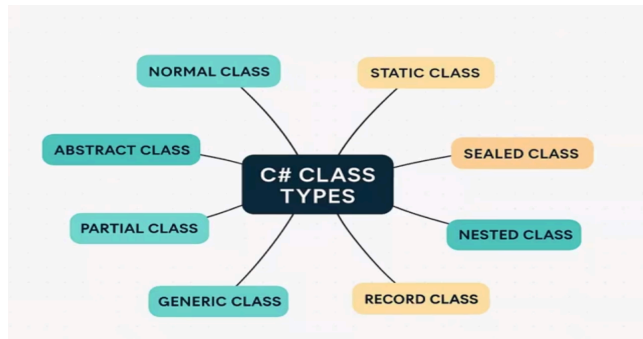
```
string s = "something";  
Console.WriteLine(s.GetType());  
Console.WriteLine(s.GetType().IsValueType);  
  
object o = a;  
Console.WriteLine(o);  
  
int b = (int)o;  
Console.WriteLine(b);
```

Access Modifier:

Modifier	Accessibility
Public	Everywhere
Private	Inside the same class only
Protected	Inside the same class and derived classes
Internal	Inside the same assembly
Protected Internal	Inside the same assembly and derived classes
Private Protected	Inside the same class and derived classes (same assembly)

Note: Access modifiers help **encapsulate** and **protect** data while enabling interaction where necessary.

Classes:



Type	Keyword / Feature	Key Rule	Where/Why to Use
Normal Class	<code>class</code> +↓	Nothing special	For general-purpose objects (like models, services, etc.).
Static Class	<code>static class</code>	Only static members, no instances	Utility/helper classes (like MathHelper, StringUtils). Shared functionality without needing an object.
Abstract Class	<code>abstract class</code>	Cannot instantiate directly	Base class when you want partial implementation. Forces derived classes to implement missing pieces . Good for shared behaviors.
Sealed Class	<code>sealed class</code>	Cannot inherit	When you want to prevent others from extending your class for security, stability, or design reasons.
Partial Class	<code>partial class</code>	Defined across files	Large classes split into multiple files for better organization (common in auto-generated code, WinForms, or big models).
Record Class	<code>record</code>	Immutable, value-based equality	When you need data-focused objects (like DTOs, API models) that care about values , not references.
Generic Class	<code>class<T></code>	Works with any type	Reusable logic for multiple data types (e.g., Collections, Services, Repositories).
Nested Class	<code>class inside class</code>	Inner class accessible via outer	Helper types that are only meaningful inside their parent class (like builder classes, or private internal helpers).

Static Class:

Definition: A static class **cannot be instantiated** and contains only **static members** (methods, properties, and fields).

✓ Key Points:

- Cannot create objects.
- Cannot **inherit** or be inherited.
- All members **must be static**.

✓ **Usage:** Used when functionality doesn't need an object, like utility/helper classes (**Math**, **Console**).

```
var sum = MathHelper.Add(10, 30);
```

Instantiated? -> Cannot **create objects** (no new keyword).


💡 **Quick Memory Trick:** "Static = Stays the Same, No Objects Needed!"

Abstract class:

Definition: An abstract class **cannot be instantiated** and serves as a **blueprint** for derived classes.

Key Points:

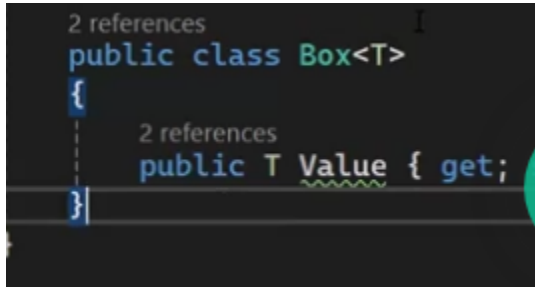
- Cannot create objects (no new keyword).
- Can have abstract methods (must be implemented in derived classes).
- Can have regular methods (can be inherited directly).

 Quick Memory Trick: "Abstract = Incomplete, Needs a Derived Class!"

Partial:

Ekta namespace/project er multiple files e same class create kra jay na.
Partial diye krte hoy. It contains all the functionality, written in different files.

Generic:



```
2 references
public class Box<T>
{
    2 references
    public T Value { get; }
}
```

property/Type will be defined in the future.

```
var box1 = new Box<int> { Value = 10};
var box2 = new Box<string> { Value = "10" };
```

Nullable in C#: (works in both value & reference data type)

`int? x = null;` Assigns null to variables. It requires `?` sign. Else, null will not be assigned.

Default in C#:

```
int defaultInt = default(int); // 0
bool defaultBool = default(bool); // false
string defaultString = default; // null
```

Enum in C#:

Fixed option thakle enum nei.

```
enum DaysOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}
```

Var in C#:

It's like `auto` in C#. The compiler determines the type of the variable

```
var name = "Alice"; // Compiler infers 'string'
var age = 25; // Compiler infers 'int'
var numbers = new List<int> { 1, 2, 3 }; // Compiler infers 'List<int>'
```

Dynamic in C#:

Type is determined at runtime. Bypass compile-time type checking.

```
dynamic value = "Hello, world!";
value = 42; // No compile-time error, type changes at runtime
```

Collection:

Stack in C #:

```
var stack = new Stack<string>();
stack.Push("First");
stack.Push(1);
stack.Push("Third");

Console.WriteLine($"Popped: {stack.Pop()}");

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

Console.WriteLine();
```

Generic [Datatype given]

U references

```
public static void Show()
{
    Console.WriteLine("--- Stack (non-generic) Example ---");

    Stack stack = new Stack();
    stack.Push("First");
    stack.Push(2);
    stack.Push(3.14);

    Console.WriteLine($"Popped: {stack.Pop()}");

    foreach (var item in stack)
    {
        Console.WriteLine(item);
    }
}
```

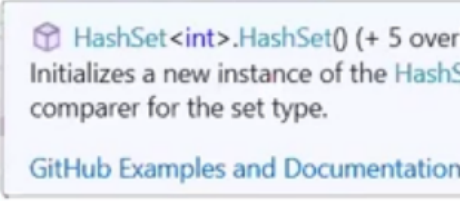
Non-Generic [Not Recommended]

HashSet in C#:

```
Console.WriteLine("--- HashSet<T> Example ---");

var uniqueNumbers = new HashSet<int> { 1, 2, 3 };
uniqueNumbers.Add(3); // Duplicate
uniqueNumbers.Add(4);

foreach (var number in uniqueNumbers)
{
    Console.WriteLine(number);
}
```



Unique value only

Dictionary in C#:(map in C++)

```
Console.WriteLine("--- Dictionary<TKey, TValue> Example ---");

var capitals = new Dictionary<string, string>
{
    {"USA", "Washington, D.C."},
    {"France", "Paris"},
    {"Japan", "Tokyo"}
};

foreach (var kvp in capitals)
{
    Console.WriteLine($"{kvp.Key}: {kvp.Value}");
}
```

ArrayList in C#: Stores objects.

```
Console.WriteLine("--- ArrayList Example ---");

ArrayList arrayList = new ArrayList();
arrayList.Add(1);
arrayList.Add("Hello");
arrayList.Add(3.14);
arrayList.Add(true);

foreach (var item in arrayList)
{
    Console.WriteLine(item);
}
```

Collections controlling thread (threadSafe)

Concurrent: (locking Mechanism)

Use `concurrent` with the collection name; it will control the thread. Use when Multi-threading may occur. Without concurrent, a lock was required. Concurrent does it by itself.

```
public static void Show()
{
    Console.WriteLine("--- ConcurrentDictionary<TKey, TValue> Example ---");

    var concurrentDict = new ConcurrentDictionary<int, string>();

    // Add key-value pairs
    concurrentDict.TryAdd(1, "One");
    concurrentDict.TryAdd(2, "Two");

    // Update existing or add new
    concurrentDict.AddOrUpdate(2, "Second", (key, oldValue) => "UpdatedTwo");
    concurrentDict.AddOrUpdate(3, "Three", (key, oldValue) => "UpdatedThree");
}
```

Immutable:

An **immutable collection** contains contents that **cannot be modified** after creation. You **cannot add, remove, or update** elements once the collection is initialized.

```
Console.WriteLine("--- ImmutableList<T> Example ---");

var list = ImmutableList.Create<int>(1, 2, 3);
var newList = list.Add(4); // old list stays unchanged

Console.WriteLine("Original List:");
foreach (var number in list)
{
    Console.WriteLine(number);
}

Console.WriteLine("New List:");
foreach (var number in newList)
{
    Console.WriteLine(number);
}
```

Delegate & event:

Delegate:

A **delegate** is a type that holds a reference to a method.

Allows callback mechanisms (method to be passed as an argument).

```
public delegate void SimpleDelegate(string message);  
1 reference  
public class BasicDelegateExample  
{  
    1 reference  
    public static void Show()  
    {  
        SimpleDelegate del = PrintMessage; ✓  
        del("Hello, World from BasicDelegate Example!");  
    }  
    1 reference  
    private static void PrintMessage(string message) ✓  
    {  
        Console.WriteLine(message);  
    }  
}
```

Handwritten annotations:

- Red arrow pointing to `SimpleDelegate`: RETURN
- Arrow pointing to `message` parameter: Parameter
- Arrow pointing to `del("Hello, World from BasicDelegate Example!");`: INPUT Assigned
- Arrow pointing to `PrintMessage` method: execute

Events:

Ekta button e press krle ki ki krbe seta hocche event and ja ja krbe shegula define thake delegate er maddhome. Events use delegates to define the notification mechanism.

MultiCast delegates:

A **multicast delegate** allows multiple methods to be assigned to a single delegate instance. When invoked, all assigned methods execute in order.

```
public delegate void Notify(string message);

Notify notify = ShowMessage;
notify += ShowUpperMessage;
//notify first references ShowMessage. notify += ShowUpperMessage; adds
another method.

notify.Invoke("Multicast Delegate Example");
//both methods execute.

private static void ShowMessage(string msg)
{
    Console.WriteLine($"Message: {msg}");
}
//ShowMessage prints the message.

private static void ShowUpperMessage(string msg)
{
    Console.WriteLine($"Uppercase Message: {msg.ToUpper()}");
}
//ShowUpperMessage converts the message to uppercase and prints it.
```

Publisher & Subscriber Model: (system design technique):

The publish-subscribe (pub-sub) model is a messaging pattern where publishers send events, and subscribers listen for them.

Also, events are stored in the queue.

Example: Order & Invoice System

Imagine an e-commerce system where:

- An order is placed (Publisher).
- The system generates an invoice (Subscriber).
- An email notification is sent (Another Subscriber).

- Class Code: [Event & Delegate Github repo \[Must check\]](#)
- Pub & Sub Model
- Delegate multiple ways [anonymous function & lambda function]
- Class Code: [C# file operations](#)[note kri nai]