**PDF 3**

**Implicitly Typed Local Variable**  
Implicitly typed local variable is a variable that can be declared without specifying the .NET type explicitly. In an implicitly typed local variable declaration, the type of the local variable is obtain from the expression used to initialize the variable. The type of the variable is inferred at compile time from the expression on the right side of initialization statement.

The “Implicitly Typed Local Variable” may only be used within class members such as methods and property. It may not be used in any element of a class where the variable could be a part of the public interface. If we use “Implicitly Typed Local Variable” for global scope then compiler will throw an error.

**Restrictions for “Implicitly Typed Local Variable”:**Now we consider some restrictions for “Implicitly Typed Local Variable” that make it unreliable variable for general purpose use.  
  
**Declaration must include an initializer:**We can’t declare a Implicitly typed Local Variable without any initialization. The declaration must include the initializing.

**Initializer must be expression:**  
  
The initializer can’t be a collection or an object, but it can be a new expression that includes an object or collection initializer.  
  
**Example:**

1. var Array\_ =
2. {
3. 10,
4. 11,
5. 12
6. }; //Error Cannot initialize an implicitly-typed local variable with an array initializer
8. var Array\_ = newint[]
9. {
10. 10,
11. 11,
12. 12
13. }; // No Error

**Implicitly typed Local Variable can’t be null:**We can’t use a null type for implicitly typed local variable

**Initializer must have same data type:**  
  
If we declare the implicitly typed local variable multiple type then each time initializers must have the same compile-time type. The implicitly-type local variable can’t be initialized with different types more than one time.

**Can’t use as parameter or return type in method:**  
  
Due to the local scope we can’t use var as parameter or as a return type for any method.

<https://www.c-sharpcorner.com/UploadFile/f0b2ed/implicitly-typed-local-variable-in-net/>

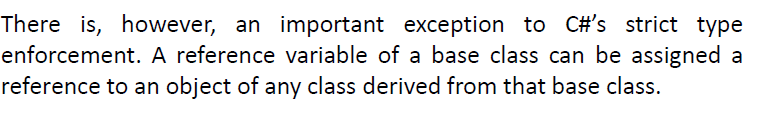
PDF 4

Class is logical and space is allocated in memory after creating object using new

Default access specifier: private

Access Specifiers:

**Usage of Access Specifiers  
  
private:**limits the accessibility of a member to within the defined type, for example if a variable or a functions is being created in a ClassA and declared as private then another ClassB can't access that.  
  
**public:**has no limits, any members or types defined as public can be accessed within the class, assembly even outside the assembly. Most DLLs are known to be produced by public class and members written in a .cs file.  
  
**internal:**internal plays an important role when you want your class members to be accessible within the assembly. An assembly is the produced .dll or .exe from your .NET Language code (C#). Hence, if you have a C# project that has ClassA, ClassB and ClassC then any internal type and members will become accessible across the classes with in the assembly.  
  
**protected:**plays a role only when inheritance is used. In other words any protected type or member becomes accessible when a child is inherited by the parent. In other cases (when no inheritance) protected members and types are not visible.  
  
**Protected internal:**is a combination of protected and internal both. A protected internal will be accessible within the assembly due to its internal flavor and also via inheritance due to its protected flavor.



Virtual keyword in base class

Override keyword in derived

Interfaces can implement other interfaces.

A class can implement one or more interfaces.

