**Constructors in c#**

**یک Special Method در کلاس که initialized Variables آن کلاس را به عهده دارد.**

**Constructor هیچ ReturnType Value ندارد.**

**هر کلاسی به یک سازنده حتما نیاز دارد در صورتی که سازنده ای را برای کلاسی تعریف نکرده باشیم**

**به صورت default برای آن کلاس implicit constructor توسط کامپایلر برای آن کلاس تعریف می شود و متغیر های کلاس را مقدار دهی اولیه (پیش فرض متغیر ها) می کند.**

**بدین ترتیب مقدار دهی اولیه متغیرها در کلاس به عهده سازنده می باشد : سازندهدر کلاس بسیار مهم است.**

**ما می توانیم Explicit Constructor در کلاس تعریف کنیم.**

**Types of Constructors in C#**

1. **Default or Parameter Less Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor**
4. **Static Constructor**
5. **Private Constructor**

##### **Default or Parameterless Constructors in C#**

اگر ctor فاقد پارامتر باشد default Constructor است. 2 نوع است

System-Defined Default Constructor : سیستم تعریف می کند

User-Defined Default Constructor کاربر تعریف میکند

Default Constructor مقادیر پیش فرض را به data Member non-staticاختصاص می دهد به

implicit Constructor توسط کامپایلر ساخته می شودو همیشه public و Parameter-Less می باشد. و چون متغیر ها را مقدار دهی اولیه می کند System-Defined Default Constructor می نامند

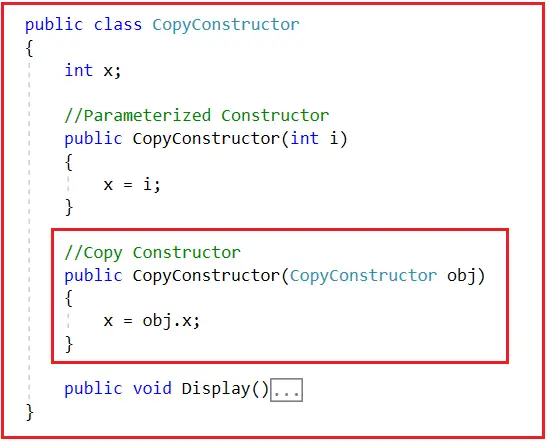
اگر بخواهیم یک custom logic را در زمان ایجاد object اجرا کنیم، آن منطق ممکن است

object initialization logic یا useful logic (منطق مفید) دیگری باشد، به عنوان یک Developer، باید Explicit Constructor در سی شارپ ارائه کنیم.

اگر بخواهیم object را به صورت dynamically )پویا (با مقادیر داده شده توسط کاربر initialize کنیم یا اگر بخواهیم each instance از یک class را با مجموعه ای متفاوت از مقادیر initialize کنیم، باید از سازنده Parameterized در سی شارپ استفاده کنیم

اگر یک constructor با پارامترها تعریف شود، آن را parameterize constructor در سی شارپ می نامیم

Copy Constructor : اگر بخواهیم Multiple instance با Same-Value داشته باشیم آنگاه باید از copy constructor استفاده کینم



در کلاس می توان چندین سازنده تعریف کرد اما باید signature متفاوت داشته باشند.

در سی شارپ امکان ایجاد یک سازنده به صورت Static وجود دارد Static Constructor :

Static Constructor مسول اولیه سازی متغیر های Static هستند

Static Constructor هرگز به صورت explicitly فراخوانی نمی شوند و Implicitly فراخوانی می شوند

Non-Static Constructor هرگز به صورت Implicitly فراخوانی نمی شود و به صورت explicitly فراخوانی می شود.

نکته : Static Constructor اولین بلوک اجرای کد در اجرای Class است.

Static Constructor را نمی توان پارامتر کرد، زیرا به صورت Implicitly اجرا میشود و فرصتی برای ارسال پارمتر نیست، پس Overloading Static Constructor امکان پذیر نیست.

Static Constructor برای Initialize the Static Members به صورت اتومات Call می شود.

در تعریف Static Constructor هیچ Access Specifiers وجود ندارد.

Static Constructor یکبار یعنی در زمان Class Loading فراخوانی می شود.

Initialize Static members در Non – Static Constructor امکان دارد اما

Initialize Non Static members در Static Constructor امکان ندارد

Private Constructor : با استفاده از Access Specifiers Private ساخته می شود در اینصورت نمی توان Object از Class در خارج از Class ایجاد کرد

Private Constructor برای ایجاد Object برای Class در همان Class استفاده می شود.

با استفاده از Private Constructor در سی شارپ می توانیم الگوی طراحی singleton را پیاده سازی کنیم.

یادآوری : **singleton چیست؟**

سینگلتون یک الگوی طراحی سازنده (Creational) است که به شما اطمینان می‌دهد از یک کلاس، تنها یک نمونه یا شی (Object) ساخته شود. بنابراین در هر زمان که استفاده از این کلاس نیاز شود، سینگلتون یک دسترسی سراسری به تنها شی‌ای که از این کلاس ساخته شده است ایجاد می‌کند و اجازه ساخت شی جدیدی را از آن کلاس نمی‌دهد.

زمانی که کلاس فقط شامل Static members باشد، باید از Private Constructor در سی شارپ استفاده کنیم.

چرا به Constructors در C# نیاز داریم؟ اگر بخواهیم instance آن Class را ایجاد کنیم، هر class نیاز به یک Constructorدارد که در آن وجود داشته باشد.

نیاز به تعریف مجدد Constructors به Explicitly چیست؟

اگر Constructors را به طور Explicitly با Parameters تعریف کنیم، هر بار که می خواهیم Instance آن Class را ایجاد کنیم، شانس initialize فیلدها یا متغیرهای class با یک مقدار جدید را خواهیم داشت.

هر زمان که class را ایجاد می کنیم یک چیز را به خاطر بسپاریم به این معنی که می توانیم از class reuse کنیم و هر زمان و هر کجا که بخواهیم می توانیم instance را ایجاد کنیم. این code reusability است.

وقتی class را Define می‌کنیم، ابتدا مشخص کنیم که آیا class Variables (منظور filed های کلاس می باشد که سطح دسترسی آنها Private است) برای اجرا به مقادیری نیاز دارند یا خیر و در صورت نیاز، constructor را به explicitly تعریف کرده و مقادیر را از constructor عبور می‌دهیم تا هر بار که instance class ایجاد می‌شود، شانس ارسال مقادیر جدید را داشته باشیم.

با کمک یک Parameterized Constructor ، می توانیم هر instance از class را با different set از مقادیر initialize کنیم

Constructor ای که parameter از نوع class را می گیرد، Copy Constructor نامیده می شود.

Static Constructor در سی شارپ فقط یک بار Call می شود

**Static vs Non-Static Constructors in C#**

1. اگر constructor ای به Explictly با استفاده از Static Modifire اعلام شود، static Constructor و بقیه Non Static Constructor است.

Static Fields / Variables -2 توسط static Constructor و

Non static Fields / Variables توسط Non-Static Constructor یا Instance

در سی شارپ Initialize می شوند

Static Constructor -3 به طور implicitly Call می شوند در حالی

که Non-static Constructor به طور Explicitly Call می شوند.

Static Constructor - 4 ، immediately از شروع اجرای یک Class اجرا می شوند و علاوه بر این، اولین بلوک کدی است که تحت یک Class اجرا می شود در حالی که

Non-Static Constructor فقط پس از ایجاد Class instance و همچنین هر بار که Class Instance ایجاد می شود، اجرا می شوند.

Static Constructor - 5 فقط یک بار اجرا می شوند در حالی که

Non-Static Constructor 0 یا n تعداد بار بسته به تعداد Class Instance ایجاد می کنیم اجرا می شوند.

1. - Non-Static Constructor را می توان Parameter کرد در حالی

Static Constructor نمی توانند هیچ Parameter داشته باشند چون به صورت call Implicitly می شود

Non-Static Constructor – 7 را می توان Overloading کرد در حالی که

Static Constructor را نمی توان Overloading کرد

Static constructor - 8 به طور Implictly تنها در صورتی تعریف می شوند که آن کلاس دارای Const Filedباشد یا در غیر این صورت آن سازنده وجود نخواهد داشت مشروط بر اینکه کلاس سازنده ایستا صریح نداشته باشد.

**Private Constructors in C#**

وقتی Constructor با استفاده از Access specifier Private تعریف می شود.

هنگامی که یک کلاس حاوی Private Constructor است و اگر کلاس هیچ Constructor Public دیگری نداشته باشد، نمی توانید یک Object برای class ، outside  از class ایجاد کنید. اما ما می توانیم برای class در همان class، object ایجاد کنیم.

Private Constructor Restricting Inheritance : Private Constructor ، Inheritance را در سی شارپ محدود می کند : یعنی اگر Private Constructor فقط در یک class داشته باشید، آن class نمی تواند ارث بری شود.

اگر می‌خواهید Inheritance را Restricting کنید، به سراغ Private Constructor نروید، در عوض، **Class Sealed** را بسازید که Class را به Inheritance از خارج یا داخل محدود می‌کند (Restricting)

اگر کلاس شما فقط Static Member دارد، Class Constructor را Private کنید. اما این اشتباه است

Private / Public Constructor برای ایجاد instance & initialize متغیرهای Non-Static یک کلاس استفاده می‌شوند.

اگر کلاس شما هیچ Non-Static Member ندارد، آیا نیازی به Constructor وجود دارد؟

پاسخ خیر است. اگر class شما **فقط** دارای Static Member است، بهتر است class Static کنید که نه تنها ایجاد instance شما را محدود می کند، بلکه inheritance را نیز محدود می کند.

در سی شارپ constructors are overloaded یعنی : ما می توانیم Constructor های متعدد با پارامترهای مختلف تعریف کنیم هم Private هم Public

چه زمانی از Private Constructors در سی شارپ استفاده کنیم؟

: 1 سازنده خصوصی برای پیاده سازی الگوی طراحی Singleton استفاده می شود

الگوی Singleton در سی شارپ چیست؟ الگوی طراحی Singleton تضمین می‌کند که تنها یک instance از یک specific Class ایجاد می‌شود و سپس دسترسی کلی ساده به آن instance را برای کل Application فراهم می‌کند.

چگونه الگوی طراحی Singleton را در سی شارپ پیاده سازی کنیم؟

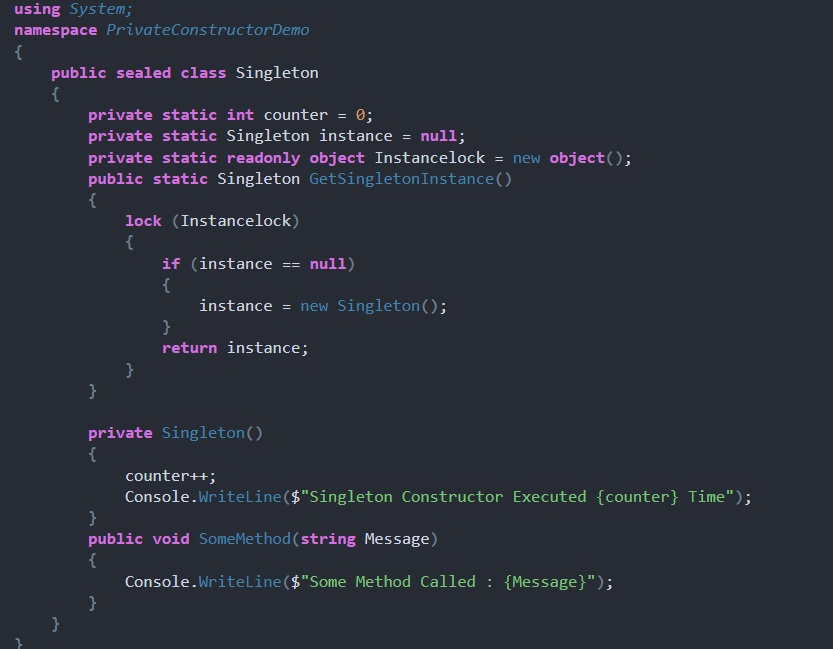
در ادامه مراحل پیاده سازی Singleton Design Pattern در سی شارپ آمده است.

شما باید فقط یک سازنده را در کلاس خود اعلام کنید و آن Private Constructor و بدون پارامتر باشد. این مورد نیاز است زیرا اجازه نمی‌دهد کلاس از خارج از کلاس نمونه‌سازی شود. این فقط از داخل کلاس نمونه برداری می کند.

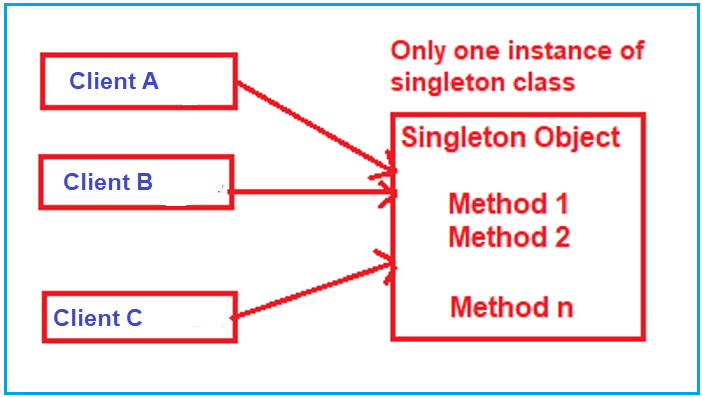
کلاس باید Sealed(مهر و موم شده)اعلام شود که تضمین می کند که نمی توان آن را به ارث برد.

شما باید یک متغیر استاتیک خصوصی بسازید که مرجعی به instance ایجاد شده تک کلاس داشته باشد.

همچنین باید یک ویژگی/روش عمومی static ایجاد کنید که نمونه تک ایجاد شده کلاس singleton را برمی گرداند.



الگوی طراحی Singleton در سی شارپ در دسته الگوهای خلاقانه (Creational Pattern Category) قرار می گیرد زیرا اطمینان می دهد Class فقط یک Instance دارد. زمانی مفید است که دقیقا به یک نمونه از یک کلاس در سراسر برنامه نیاز داریم.در سناریو هایی که چندین Object نیاز به یک منبع مشترک دارند مانند تنظیمات پیکربندی – دسترسی به سیستم فایل یا مدیریت اتصال به یک پایگاه داده.



[**https://dotnettutorials.net/lesson/singleton-design-pattern/**](https://dotnettutorials.net/lesson/singleton-design-pattern/)

**Destructors in C#**

به گفته MSDN، Destructorهایی که در سی شارپ Finalizers نیز نامیده می‌شوند، برای انجام هرگونه پاکسازی نهایی ضروری زمانی که یک instance class توسط garbage collector می‌شود، استفاده می‌شود.

Destructor نیز نوع خاصی از متد موجود در یک کلاس است درست مانند یک constructor، که نامی مشابه با نام class دارد. از طرف دیگر، زمانی که object class ، destroyedمی شود. Destructor در سی شارپ به طور implicitly فراخوانی می شود.برای تمایز بین constructor و Destructor از نماد tilde (~) قبل از destructor method استفاده می شود.

Destructor نمی تواند پارامتری داشته باشد.

چه زمانی یک متد Destructor در سی شارپ Called می شود؟

هنگامی که class object از بین می رود، یک Destructor Method به طور خودکار توسط garbage collector فراخوانی می شود.

چه زمانی object یک کلاس در سی شارپ destroyed می شود؟

در پایان اجرای برنامه (end of the program execution) ، هر object ای که با برنامه مرتبط است توسط garbage collector از بین می رود.

Implicit calling of the garbage collector زمانی در اواسط اجرای برنامه اتفاق می‌افتد، به شرطی که حافظه پر باشد، بنابراین garbage collector ، objects استفاده‌نشده application را شناسایی کرده و آنها را از بین می‌برد.

Explicit calling of the garbage collector نیز می توان در اواسط اجرای برنامه توسط برنامه نویس با کمک عبارت “Collect()” انجام داد تا در صورت وجود objects استفاده نشده مرتبط با برنامه در وسط اجرای برنامه از بین برود.

Destructors (or Finalizers) را نمی توان در Staruct تعریف کرد فقط در کلاس ها استفاده می شود.

در یک کلاس، ما فقط می توانیم یک Destructor (یا Finalizer) تعریف کنیم.

Destructor را نمی توان به طور Explicit نامید. آنها به طور خودکار توسط garbage collector فراخوانی می شوند. حداکثر کاری که می‌توانیم انجام دهیم این است که می‌توانیم با فراخوانی متد ()GC.Collect از garbage collector درخواست کنیم تا Destructor را اجرا کند.

همانطور که می بینید، Destructor به طور ضمنی متد Finalize را در کلاس پایه Object فراخوانی می کند

نکته 1 : از Destructors خالی نباید در برنامه ما استفاده شود. هنگامی که یک کلاس حاوی یک Destructors است، یک ورودی برای آن Destructors در صف Finalize ایجاد می شود. این صفی است که توسط garbage collector پردازش می شود. هنگامی که Garbage Collector صف Finalize را پردازش می کند، هر Destructors را call می کند. بنابراین، Destructors غیرضروری، از جمله Destructors خالی، تخریب‌کننده‌هایی که فقط کلاس پایه را Destructors می‌خوانند، یا Destructors که فقط متدهای منتشر شده مشروط را فراخوانی می‌کنند، باعث از دست دادن بی‌ضروری performance می‌شوند.

نکته 2 : ما به عنوان یک Developer، هیچ کنترلی روی زمان فراخوانی و اجرای Destructor نداریم. garbage collector در دات‌نت فریم‌ورک است که تصمیم می‌گیرد چه زمانی با Destructors تماس بگیرد. garbage collector به صورت دوره ای object هایی را که دیگر توسط برنامه استفاده نمی شود بررسی می کند. اگر یک object را واجد شرایط برای تخریب در نظر بگیرد، سپس Destructor (در صورت وجود) را call می کند و حافظه مورد استفاده برای ذخیره object را بازیابی می کند. همچنین می‌توان با فراخوانی متد GC.Collect، جمع‌آوری زباله را اجباری کرد، اما در بیشتر مواقع باید از این فراخوانی اجتناب کرد زیرا ممکن است مشکلاتی در عملکرد ایجاد کند.

پس چه زمانی باید از Destructor استفاده کنیم؟

به طور کلی، به عنوان توسعه دهندگان C#.NET، نیازی نیست که بیشتر نگران مدیریت حافظه باشیم. این به این دلیل است که .NET garbage collector به طور implicitly

Allocate && Deallocate حافظه را برای Objects ما مدیریت می کند.

با این حال، زمانی که برنامه ما با منابع مدیریت نشده مانند ویندوز، فایل ها و اتصالات شبکه کار می کند، باید از یک Destructor برای آزاد کردن حافظه آن منابع مدیریت نشده استفاده کنیم. هنگامی که object واجد شرایط تخریب باشد، garbage collector روش Finalize شی را اجرا می‌کند.

در حال حاضر در مدیریت حافظه در کلاس MemoryCacheManager از GC.SuppressFinalize(this); استفاده می کنیم.

Explicit Release منابع با استفاده از Dispose Pattern در سی شارپ:

اگر برنامه ما از یک منبع خارجی گران قیمت استفاده می کند، توصیه می شود قبل از اینکه garbage collector object را آزاد کند، یک explicit way برای آزاد کردن (release)منابع ارائه دهیم <= برای انتشار منبع، توصیه می‌شود یک روش Dispose از رابط IDisposable پیاده‌سازی شود که پاکسازی لازم را برای شی انجام می‌دهد.

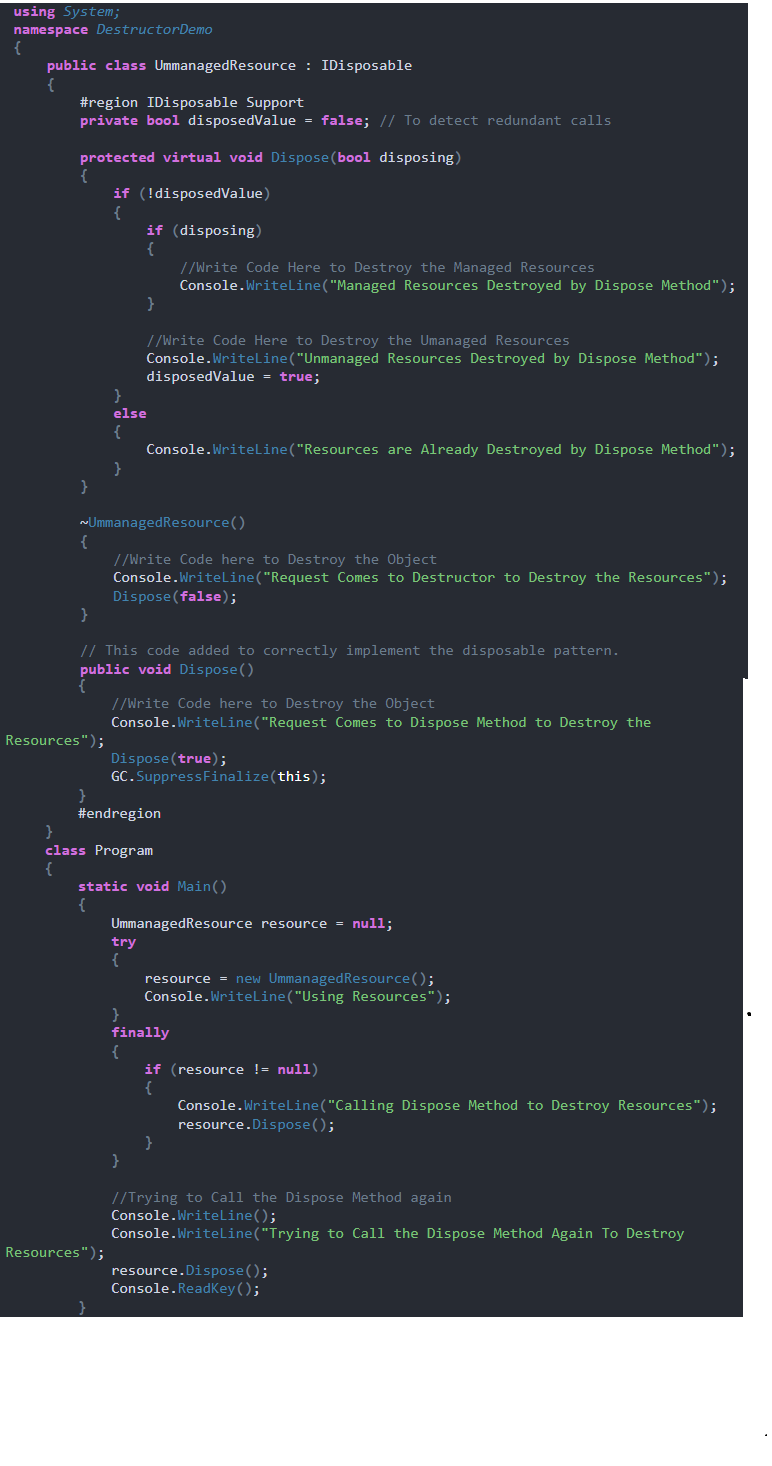
درک Pattern Dispose در سی شارپ:

disposedValue **:** handle redundant callما درخواست destroy منابع را داده ایم و اگر مجدداً درخواست destroy object را داشته باشیم، این فیلد را بررسی می کند و تشخیص می دهد که آیا call اضافی است یا خیر

~UmmanagedResource(): به‌طور پیش‌فرض، می‌بینید که کد تخریب‌کننده یا نهایی‌کننده نظر داده شده است. اما اگر می‌خواهید منابع را از طریق Destructor از بین ببرید، باید این destructor یا finalizer را از کامنت بردارید. اما، به عنوان بخشی از این تخریبگر، باید کد پاکسازی را برای منابع مدیریت شده بنویسید و سپس باید متد Dispose(false) را برای پاکسازی منابع مدیریت نشده فراخوانی کنید.

Dispose(bool disposing): باید منطق واقعی را بنویسیم تا منابع مدیریت شده و مدیریت نشده درون این روش را از بین ببریم.

Dispose(): این روشی است که باید از کد خود فراخوانی کنیم تا منابع مدیریت شده و مدیریت نشده را پاک کنیم.



**Garbage Collection in .NET Framework**

یک Small Routine یا Background Process Thread سعی می کند تشخیص دهد چه اشیایی در حال حاضر توسط برنامه استفاده نمی شوند و حافظه آن اشیا را از بین می برد.

بنابراین، Garbage Collector چیزی نیست جز ویژگی ارائه شده توسط CLR که به ما کمک می کند اشیاء مدیریت شده استفاده نشده را تمیز یا از بین ببریم.

Garbage Collection (GC) در .NET Framework یک سیستم مدیریت حافظه خودکار است که به مدیریت allocation و release حافظه در برنامه های شما کمک می کند

Managed and Unmanaged Objects in .NET Framework:

وقتی از این "EXE ها" در برنامه خود استفاده می کنید، اینها توسط CLR اجرا نمی شوند. حتی اگر شما این "EXE" را در برنامه های dot net اجرا کنید، آنها تحت محیط خود اجرا می شوند.

کدهایی که تحت کنترل کامل CLR اجرا می شوند در چارچوب دات نت کد مدیریت شدهManaged Code نامیده می شوند کد (EXE، Web App) که تحت کنترل CLR اجرا نمی شود، کد مدیریت نشده نامیده می شود.

Managed Objects:

Managed Objects یا اشیاء مدیریت شده بر روی پشته مدیریت شده Managed Heap تخصیص داده می شوند و توسط Garbage Collector دات نت (GC) کنترل می شوند.

این اشیا معمولاً نمونه هایی از classes و structures هستند که در NET تعریف شده اند.

GC به طور خودکار حافظه اشیاء مدیریت شده را مدیریت می کند. حافظه را برای این اشیاء اختصاص داده و آزاد می کند

Unmanaged Objects:

اشیاء مدیریت نشده اشیایی هستند که حافظه آنها توسط NET GC مدیریت نمی شود.

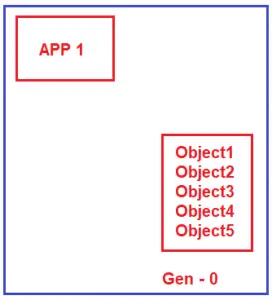
اینها معمولاً اشیایی هستند که با استفاده از کد بومی (native code)، مانند تماس با Windows API یا با استفاده از زبان‌هایی مانند C یا C++ تخصیص داده می‌شوند.

developerمسئول تخصیص و آزادسازی حافظه برای اشیاء مدیریت نشده است.

بیایید بفهمیم Generation )نسل‌های( Garbage Collector چیستند؟

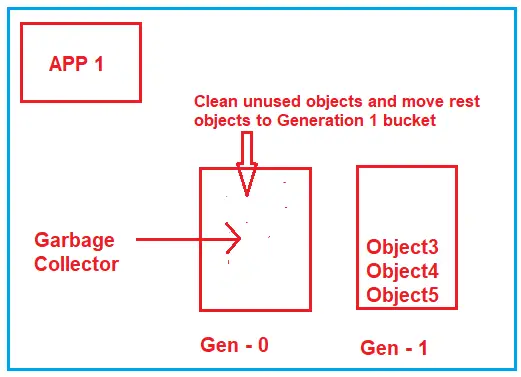
سه Generation وجود دارد : Generation 0, Generation 1, and Generation 2.

هر زمان که Objects جدیدی (Fresh Object) ایجاد می‌شوند، به سطلی به نام نسل 0 منتقل می‌شوند

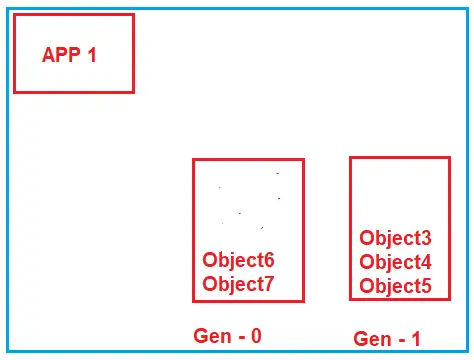


می دانیم Garbage Collector، به طور مداوم به عنوان یک background process thread اجرا می‌شود تا بررسی کند که آیا Objects مدیریت‌شده‌ی بلااستفاده‌ای وجود دارد یا خیر، تا با پاکسازی آن اشیاء، حافظه را بازیابی کند

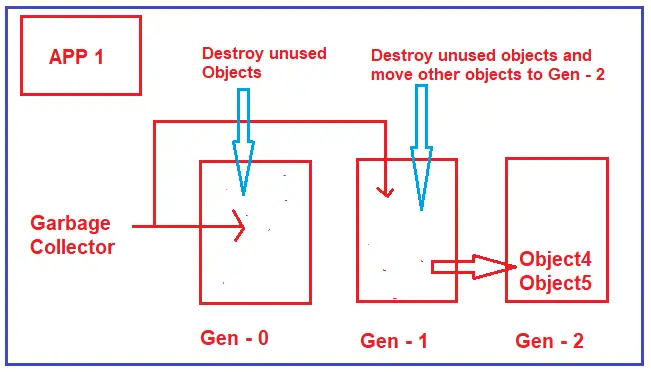
حال، فرض کنید برنامه به دو شیء (Object1 و Object2) نیاز ندارد. بنابراین، Garbage Collector این دو شیء (Object1 و Object2) را از بین می‌برد و حافظه را از Generation 0 bucket )سطل نسل(0 بازیابی می‌کند. اما برنامه هنوز به سه شیء باقی‌مانده (Object3، Object4 و Object5) نیاز دارد. بنابراین، Garbage Collector آن سه شیء را پاکسازی نمی‌کند. Garbage Collector آن سه شیء مدیریت‌شده (Object3، Object4 و Object5) را همانطور که در تصویر زیر نشان داده شده است، به Generation 1 bucket (سطل نسل1 ) منتقل می‌کند.



فرض کنید برنامه شما دو شیء جدید دیگر (Object6 و Object7) ایجاد می‌کند. به عنوان اشیاء جدید، آنها باید در Generation 0 bucket (سطل نسل ۰ )ایجاد شوند، همانطور که در تصویر زیر نشان داده شده است.



حالا، دوباره، Garbage Collector اجرا می‌شود و به سراغ در Generation 0 bucket (سطل نسل ۰ )می‌رود و بررسی می‌کند که کدام اشیاء استفاده شده‌اند. فرض کنید هر دو شیء (Object6 و Object7) توسط برنامه استفاده نشده باشند، بنابراین هر دو شیء را حذف کرده و حافظه را آزاد می‌کند. حالا، به سراغ Generation 1 bucket (سطل نسل1 ) می‌رود و بررسی می‌کند که کدام اشیاء استفاده نشده‌اند. فرض کنید برنامه هنوز به Object4 و Object5 نیاز دارد در حالی که به object3 نیازی ندارد. بنابراین، کاری که Garbage Collector انجام می‌دهد این است که Object3 را از بین می‌برد، حافظه را آزاد می‌کند و Object4 و Object5 را به Generation 2 bucket (سطل نسل2 ) منتقل می‌کند، همانطور که در تصویر زیر نشان داده شده است.



Generations چیستند؟

Generations چیزی نیستند جز اینکه مشخص می‌کنند Objects چه مدت در حافظه باقی می‌مانند. حال، سوالی که باید به ذهنتان خطور کند این است که چرا به نسل‌ها نیاز داریم؟ چرا سه نوع نسل مختلف داریم؟

اگر یک شیء در سطل‌های Generation 2 باشد، Garbage Collector بازدیدهای کمتری از این سطل انجام می‌دهد، به چه معناست؟ دلیل آن این است که اگر یک شیء به Generation 2 منتقل شود، مدت بیشتری در حافظه باقی می‌ماند. دلیلی برای رفتن و بررسی مجدد آنها وجود ندارد.

بنابراین، به عبارت ساده، می‌توانیم بگوییم که Generationهای 0، 1 و 2 به افزایش عملکرد Garbage Collector کمک می‌کنند. هرچه اشیاء در Gen 0 بیشتر باشند، عملکرد بهتر می‌شود و حافظه به شیوه‌ای بهینه‌تر مورد استفاده قرار می‌گیرد.

نسل 0 : بیشتر از نسل های دیگر جمع آوری می شود.

اکثر اشیا در این نسل برای garbage collection بازیابی می شوند زیرا عمر کوتاهی دارند

فرآیند جمع آوری سریع است زیرا بخش کوچکی از هیپ را اشغال می کند

مثال : متغیرهای موقت، اشیاء کوچک و اشیاء که به طور خلاصه در متدها استفاده می‌شوند.

نسل ۱ (Gen 1) به عنوان یک بافر بین اشیاء با عمر کوتاه در نسل ۰ و اشیاء با عمر طولانی در نسل ۲ عمل می‌کند.

نسل ۱ کمتر از نسل ۰ garbage collection می‌شود

اشیایی که چندین چرخه GC را پشت سر می‌گذارند اما در طول عمر برنامه استفاده نمی‌شوند

نسل ۲ : اشیاء با عمر طولانی: نسل ۲ (Gen 2) شامل اشیاء با عمر طولانی است

کمترین میزان جمع‌آوری

مثال‌ها: اشیاء استاتیک، اشیاء مرتبط با عمر برنامه و اشیاء بزرگی که به منابع حافظه قابل توجهی نیاز دارند.

وقتی یک destructor در کلاس خود تعریف می کنید Garbage Collector قبل از Destroy Object از class می پرسد آیا destructor دارد یا خیر،اگر داشت آن را به سطل بعدی منتقل می کند

Object دارای Destructor را در آن لحظه حتی اگر استفاده نشده باشد ، پاک نمی کند منتظر می ماند تا Destructor اجرا شود و سپس می رود و Object را پاک می کند.به همین دلیل اشیا بیشتری در سطح 1و2و هستند.

بنابراین، اگر کد پاکسازی را در مخرب خود می‌نویسید، اشیاء را در نسل ۱ و نسل ۲ ایجاد خواهید کرد، به این معنی که از حافظه به درستی استفاده نمی‌کنید.

این مشکل را می‌توان با استفاده از چیزی به نام الگوی Finalized Dispose برطرف کرد

برای پیاده‌سازی این، کلاس شما باید رابط IDisposable را پیاده‌سازی کند و پیاده‌سازی متد Dispose را فراهم کند.

در داخل متد Dispose، باید کد پاکسازی را برای اشیاء مدیریت نشده بنویسید و در نهایت، باید متد GC.SuppressFinalize(true) را با ارسال مقدار ورودی true فراخوانی کنید. این متد هر نوع مخربی را سرکوب می‌کند و فقط اشیاء را پاکسازی می‌کند



تفاوت بین Finalize and Dispose in C# در سی شارپ :

Finalize و Dispose هر دو روشی هستند که برای آزادسازی منابع استفاده می‌شوند.

متد Finalize در سی‌شارپ:

هدف: روش Finalize برای عملیات پاکسازی قبل از garbage Collector یک object استفاده می‌شود. معمولاً برای آزادسازی Unmanage Resource که Object در اختیار دارد، بازنویسی می‌شود. جمع‌آوری‌کننده زباله، روش Finalize را به طور خودکار فراخوانی می‌کند.

کنترل: شما Finalize را مستقیماً فراخوانی نمی‌کنید. توسط garbage Collector فراخوانی می‌شود.

غیر قطعی: زمان دقیق فراخوانی Finalize بسته به برنامه garbage Collector ، غیر قطعی است.

وراثت: روش Finalize از کلاس Object به ارث رسیده است. در صورت بازنویسی، همیشه باید روش Finalize کلاس پایه خود را فراخوانی کند تا اطمینان حاصل شود که همه منابع به درستی آزاد می‌شوند.

متد Dispose: بخشی از interface IDisposable است و برای آزادسازی قطعی منابع مدیریت‌شده و مدیریت‌نشده پیاده‌سازی شده است.

کنترل: برخلاف Finalize، Dispose به طور صریح در کد شما فراخوانی می‌شود. معمولاً زمانی که استفاده از یک شیء تمام می‌شود. این امر امکان آزادسازی فوری منابع را فراهم می‌کند.

قطعی: Dispose روشی قطعی برای آزادسازی منابع ارائه می‌دهد، به این معنی که دقیقاً می‌دانید منابع چه زمانی آزاد می‌شوند.

الگو: هنگام پیاده‌سازی Dispose، معمولاً از الگوی dispose پیروی می‌شود که شامل فراخوانی finalizer (GC.SuppressFinalize(this)) است تا از فراخوانی Finalize توسط garbage collector در صورتی که Dispose قبلاً فراخوانی شده باشد، جلوگیری شود.

تفاوت‌های کلیدی بین Finalize و Dispose در سی‌شارپ:

زمان‌بندی: Finalize توسط garbage collector به صورت غیرقطعی فراخوانی می‌شود، در حالی که Dispose به طور صریح در یک نقطه مشخص در برنامه فراخوانی می‌شود.

منابع: Finalize معمولاً برای منابع مدیریت نشده استفاده می‌شود، در حالی که Dispose می‌تواند برای منابع مدیریت شده و مدیریت نشده استفاده شود.

کنترل: Dispose در مقایسه با Finalize کنترل بیشتری بر مدیریت منابع به شما می‌دهد.

بهترین شیوه‌ها:

پیاده‌سازی Dispose برای امکان پاکسازی قطعی منابع.

از یک finalizer (متد Finalize) فقط برای پاکسازی منابع مدیریت نشده‌ای که در یک دسته امن قرار نگرفته‌اند و زمانی که هیچ تضمینی برای فراخوانی Dispose وجود ندارد، استفاده کنید.

در متد Dispose، GC.SuppressFinalize را فراخوانی کنید تا از فراخوانی Finalize توسط garbage collector، در صورتی که قبلاً dispose شده باشد، جلوگیری شود.

از الگوی dispose پیروی کنید، به خصوص اگر کلاس شما دارای منابع مدیریت نشده است.

**Access Specifiers in C#**

Access Specifiers در سی‌شارپ چیستند؟

هر کلمه کلیدی که ما استفاده می‌کنیم مانند private، public، protected، virtual، sealed، partial، abstract، static، base و غیره، اصلاح‌کننده (Modifier) ​​نامیده می‌شود.

مشخصه‌های دسترسی انواع خاصی از Access Specifiers هستند که با استفاده از آنها می‌توانیم دامنه یک نوع(scope of a type) و اعضای (members) آن را تعریف کنیم.

scope of a type برای : Class, Interface, Structs, Delegate, Enum, etc.

scope of Member : Variables, Properties, Constructors, and Methods

سی‌شارپ از 6 نوع Access Specifiers پشتیبانی می‌کند.

1. **Private**
2. **Public**
3. **Protected**
4. **Internal**
5. **Protected Internal**
6. **Private Protected**

scope of Members می‌توانند هر 6 Access Specifiers ​​داشته باشد

. Scope Of a Type فقط می‌تواند Access Specifiers (internal, public) داشته باشند.

اگر هیچ نوعی را مشخص نکرده باشیم، برای type، internal access specifier

​​و برای Type Members، private access specifier​​ خواهد بود

اسمبلی‌ها در دات نت فریم ورک چیستند؟

طبق MSDN، اسمبلی‌ها بلوک سازنده برنامه‌های دات نت (building block of .NET) هستند

آنها واحد اساسی استقرار را تشکیل می‌دهند. به عبارت ساده، می‌توان گفت که اسمبلی چیزی جز یک کد دات نت از پیش کامپایل شده نیست که می‌تواند توسط CLR (Common Language Runtime) اجرا شود.

برای یک console application، اسمبلی EXE و برای پروژه class library ، اسمبلی DLL است.

Access Specifiers or Access Modifiers ، scope of the type members را تعریف می کند.

scope of the type members ها (مثال اعضای یک کلاس) به شرح زیر است :

در کلاس

کلاس مشتق شده در همان اسمبلی

کلاس غیر مشتق شده در همان اسمبلی

کلاس مشتق شده در سایر اسمبلی‌ها

کلاس غیر مشتق شده در سایر اسمبلی‌ها

Private Access Specifier or Modifier :

وقتی یک عضو نوع (variable, property, method, constructor, etc) را به صورت Private تعریف می‌کنیم، می‌توانیم فقط در کلاس به آن عضو دسترسی داشته باشیم. از خارج از کلاس نمی‌توانیم به آنها دسترسی داشته باشیم.

بنابراین scope of the private member in C#.NET :

1. **With the Class: YES**
2. **Derived Class in Same Assembly: NO**
3. **Non-Derived Class in Same Assembly: NO**
4. **Derived Class in Other Assemblies: NO**
5. **Non-Derived Class in Other Assemblies: NO**

Public Access Specifiers or Access Modifiers :

وقتی یک عضو نوع (متغیر، ویژگی، متد، سازنده و غیره) را به صورت عمومی (public) تعریف می‌کنیم، می‌توانیم از هر جایی به آن عضو دسترسی داشته باشیم. این یعنی هیچ محدودیتی برای اعضای عمومی وجود ندارد.

1. **With the Class: YES**
2. **Derived Class in Same Assembly: YES**
3. **Non-Derived Class in Same Assembly: YES**
4. **Derived Class in Other Assemblies: YES**
5. **Non-Derived Class in Other Assemblies: YES**

Protected Access Specifier or Access Modifier :

اعضای محافظت‌شده هم در کلاس والد (یعنی نوع حاوی) و هم در کلاس‌های فرزند/مشتق‌شده (کلاس‌های مشتق‌شده از نوع حاوی) در دسترس هستند.

1. **With the Class: YES**
2. **Derived Class in Same Assembly: YES**
3. **Non-Derived Class in Same Assembly: NO**
4. **Derived Class in Other Assemblies: YES**
5. **Non-Derived Class in Other Assemblies: NO**

Internal Access Specifier or Access Modifier :

هر زمان که یک عضو با استفاده از مشخصه دسترسی داخلی در سی‌شارپ تعریف شود، در هر جایی از اسمبلی حاوی آن در دسترس خواهد بود. دسترسی به یک عضو داخلی از خارج از اسمبلی حاوی آن، یک خطای زمان کامپایل محسوب می‌شود.

1. **With the Class: YES**
2. **Derived Class in Same Assembly: YES**
3. **Non-Derived Class in Same Assembly: YES**
4. **Derived Class in Other Assemblies: NO**
5. **Non-Derived Class in Other Assemblies: NO**

##### Protected Internal Access Specifier or Access Modifier  :

ترکیبی Protected and Internal است

هم اعضا می‌توانند درون کلاس‌های مشتق‌شده قابل دسترسی باشند و هم درون همان اسمبلی قابل دسترسی باشند.

1. **With the Class: YES**
2. **Derived Class in Same Assembly: YES**
3. **Non-Derived Class in Same Assembly: YES**
4. **Derived Class in Other Assemblies: YES**
5. **Non-Derived Class in Other Assemblies: NO**

##### Private Protected Access Specifier or Access Modifier :

در داخل کلاس و در داخل کلاس مشتق شده از همان اسمبلی قابل دسترسی هستند، اما از اسمبلی دیگری قابل دسترسی نیستند.

1. **With the Class: YES**
2. **Derived Class in Same Assembly: YES**
3. **Non-Derived Class in Same Assembly: NO**
4. **Derived Class in Other Assemblies: NO**
5. **Non-Derived Class in Other Assemblies: NO**



**Access Specifiers with Type** :

ما می‌توانیم از هر 6 مشخص‌کننده دسترسی با اعضای نوع در سی‌شارپ استفاده کنیم، اما نوع فقط دو مشخص‌کننده دسترسی یعنی Public & Internal را مجاز می‌داند. استفاده از مشخص‌کننده‌های دسترسی Private، Protected، protected Internal و private Protected با نوع‌ها، یک خطای زمان کامپایل ایجاد می‌کند. که بیان می‌کند عناصر تعریف‌شده در یک فضای نام نمی‌توانند به طور صریح به عنوان Private، Protected، protected Internal و private Protected اعلام شوند.

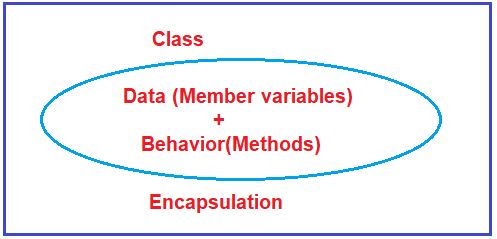
نکته: نکته‌ای که باید به خاطر داشته باشید این است که اگر می‌خواهید فقط از داخل same assembly به کلاس دسترسی داشته باشید، باید کلاس را به صورت intrenal تعریف کنید و اگر می‌خواهید هم از same assembly و هم از other assemblies به کلاس دسترسی داشته باشید، باید کلاس را به صورت public تعریف کنید.

**Encapsulation in C#**

اصل کپسوله‌سازی در سی‌شارپ چیست؟

طبق MSDN، encapsulation حالت داخلی و functionality یک object را پنهان می‌کند و فقط از طریق مجموعه‌ای public از توابع امکان دسترسی به آن را فراهم می‌کند

Principal encapsulation تضمین می‌کند که حالت و رفتار یک واحد (یعنی کلاس، رابط، ساختار و غیره) مستقیماً از واحدهای دیگر (یعنی کلاس، رابط، ساختار و غیره) قابل دسترسی نباشد

.

Encapsulation principal در سی‌شارپ بسیار شبیه به کپسول است. همانطور که یک کپسول، داروی خود را درون خود قرار می‌دهد، به همان روش در سی‌شارپ، Encapsulation principal ، حالت (Variables) and Behaviour (Methods) را در یک واحد واحد به نام کلاس، enum، رابط و غیره قرار می‌دهد.

Encapsulation در سی‌شارپ، یعنی واحدها (کلاس، رابط، enums، structs و غیره) تمام data Member و Method Member عضو خود را درون خود جای می‌دهد.

مثال دیگر کپسوله‌سازی در دنیای واقعی می‌تواند کیف مدرسه یا محل کار شما باشد. کیف حاوی چیزهای مختلفی مانند خودکار، مداد، دفترچه یادداشت و غیره است. برای دریافت هر چیزی، باید آن کیف را باز کنید. به طور مشابه، در سی‌شارپ، یک واحد کپسوله‌سازی شامل داده‌ها و رفتارهای خود در درون خود است و برای دسترسی به آنها، به یک شیء از آن واحد نیاز دارید.

اگر یک رابط، ساختار یا enum ایجاد کنید، این نیز نمونه‌ای از کپسوله‌سازی است.

بزرگترین مزیت Encapsulation ، Data Hiding (پنهان‌سازی داده‌ها) است. این بدان معناست که با استفاده از Encapsulation ، می‌توانیم به Data Hiding دست یابیم

Data Hiding داده‌ها در سی‌شارپ چیست؟ فرآیندی است که در آن internal data را از دنیای خارج پنهان می‌کنیم. هدف از Data Hiding محافظت از داده‌ها در برابر سوءاستفاده توسط دنیای خارج است

بدون Encapsulation Principal ، نمی‌توانیم به Data Hiding دست یابیم.

به عبارت ساده، می‌توانیم بگوییم که فرآیند تعریف یک Class با Hiding اعضای داده داخلی آن از خارج از کلاس و دسترسی به آن اعضای داده داخلی فقط از طریق متدهای در معرض دید عمومی (متدهای setter و getter) یا ویژگی‌هایی با proper validations (اعتبارسنجی مناسب)، Encapsulation نامیده می‌شود.

**چگونه می‌توانیم پنهان‌سازی داده‌ها یا کپسوله‌سازی داده‌ها را در سی‌شارپ پیاده‌سازی کنیم؟**

با تعریف Variables به صورت Private (برای محدود کردن دسترسی مستقیم آنها از خارج از کلاس)

با تعریف یک جفت Method یا Property عمومی setter و getter برای access Private Variable از خارج از class.

نکته : اگر دسترسی مستقیم به variables یک class را فراهم کنیم، نمی‌توانیم data را قبل از ذخیره آنها در variables یا هنگام بازیابی data ذخیره شده در variable، validate کنیم.

با Encapsulation در واقع ما از داده ها محافظت می کنیم.

Encapsulation و Data Hiding در سی شارپ با استفاده از Access Specifiers پیاده سازی می شود.

مزایای ارائه دسترسی به متغیر از طریق متدهای Setter و Getter : می‌توانیم داده‌های داده شده توسط کاربر را قبل از ذخیره مقدار در متغیر، Validate کنیم

بنابراین، دلیل اصلی استفاده از hiding یا Encapsulation داده‌ها، امنیت است.

**مزایای کپسوله‌سازی در سی‌شارپ:**

**Data protection**: می‌توانید داده‌ها را قبل از ذخیره در متغیر، اعتبارسنجی کنید.

**Achieving Data Hiding**: کاربر هیچ اطلاعی از پیاده‌سازی داخلی کلاس نخواهد داشت.

**Security**: اصل کپسوله‌سازی به ایمن‌سازی کد ما کمک می‌کند زیرا تضمین می‌کند که واحدهای دیگر (کلاس‌ها، رابط‌ها و غیره) نمی‌توانند مستقیماً به داده‌ها دسترسی داشته باشند.

**Flexibilit**: اصل کپسوله‌سازی در سی‌شارپ کد ما را انعطاف‌پذیرتر می‌کند و به برنامه‌نویس اجازه می‌دهد تا به راحتی کد را تغییر دهد یا به‌روزرسانی کند.

**Control**: اصل کپسوله‌سازی کنترل بیشتری بر داده‌های ذخیره شده در متغیرها می‌دهد. به عنوان مثال، می‌توانیم داده‌ها را با اعتبارسنجی اینکه آیا داده‌ها به اندازه کافی خوب هستند که در متغیر ذخیره شوند، کنترل کنیم.

**Encapsulationیکی از چهار principal اساسی برنامه‌نویسی شی‌گرا (OOP) است**. این به دسته‌بندی داده‌ها (حالت) و روش‌ها (رفتارها) که روی داده‌ها عمل می‌کنند در یک واحد واحد و محدود کردن دسترسی به برخی از اجزای شیء اشاره دارد.

Abstraction

Abstarction در سی شارپ یکی از اصول اساسی OOP است که به عنوان یک اصل پشتیبان عمل می‌کند.

این بدان معناست که اصل Abstarction در سی شارپ تضمین می‌کند که هر سه اصل دیگر (Encapsulation، Polymorphismو inhertance) با هم کار می‌کنند تا شکل نهایی پروژه را ایجاد کنند.

Abstraction در سی شارپ چیست؟

فرآیند Representation ویژگی‌های ضروری بدون درج جزئیات پس‌زمینه، Abstraction نامیده می‌شود.

به عبارت ساده، می‌توان گفت که این فرآیند تعریف یک Class با ارائه جزئیات لازم برای فراخوانی عملیات Object (یعنی متدها) با پنهان کردن جزئیات Implementation آن است. این کار در سی شارپ Abstraction نامیده می‌شود.

به این معنی که ما باید آنچه را که ضروری و اجباری است، آشکار کنیم و چیزهای غیرضروری را از دنیای خارج پنهان کنیم.

Abstraction به شما امکان می‌دهد بیشتر بر روی کاری که یک شیء انجام می‌دهد تمرکز کنید تا نحوه انجام آن.

به عبارت ساده، می‌توانیم بگوییم که Abstraction به معنای ارائه جزئیات لازم برای استفاده از خدمات یا قابلیت‌ها است، اما نحوه پیاده‌سازی داخلی خدمات که باید پنهان کنید چیست.

این ممکن است باعث مشکلات امنیتی شود. ما نباید جزئیات پیاده‌سازی خود را در معرض دید دیگران قرار دهیم.

چگونه Abstraction Principle را در سی شارپ پیاده‌سازی کنیم؟

در سی شارپ، می‌توانیم اصل انتزاع OOPs را به دو روش پیاده‌سازی کنیم. این روش‌ها به شرح زیر هستند:

استفاده از **Interface**

استفاده از کلاس‌های **Abstraction** و Method **Abstraction**

هم کلاس‌های **Interface** و **Abstraction** و هم Method **Abstraction** ، مکانیسمی را برای پنهان کردن جزئیات پیاده‌سازی با نمایش سرویس‌ها ارائه می‌دهند.

با استفاده از اصل Encapsulation ، می‌توانیم از داده‌های خود محافظت کنیم، یعنی از خارج از کلاس، هیچ‌کس نمی‌تواند مستقیماً به داده‌ها دسترسی پیدا کند. ما داده‌ها را از طریق متدها و ویژگی‌های در معرض دید عموم در معرض نمایش قرار می‌دهیم. مزیت این است که می‌توانیم داده‌ها را قبل از ذخیره و بازگرداندن آنها Validate کنیم. از سوی دیگر، با استفاده از چگونه Abstraction Principle ، ما فقط سرویس‌ها را در معرض نمایش قرار می‌دهیم تا کاربر بتواند از سرویس‌ها استفاده کند، اما نحوه پیاده‌سازی سرویس‌ها/متدها از کاربر پنهان است. کاربر هرگز نخواهد دانست که روش چگونه پیاده‌سازی شده است.

با اصل Encapsulation ، ما اعضای داده و توابع عضو را در یک واحد واحد به نام class, interface, enum و غیره گروه‌بندی می‌کنیم. از سوی دیگر، با Abstraction Principle ، ما interface یا class **Abstraction** را در معرض نمایش کاربر قرار می‌دهیم و جزئیات پیاده‌سازی را پنهان می‌کنیم، یعنی اطلاعات کلاس فرزند را پنهان می‌کنیم.

ما می‌توانیم Encapsulation را با اعلام اعضای داده به عنوان خصوصی (private) و افشای اعضای داده فقط از طریق method و properties Private با Validate مناسب پیاده‌سازی کنیم. از سوی دیگر، می‌توانیم **Abstraction** را از طریق abstract classes and interfaces پیاده‌سازی کنیم.

در **Abstraction** ، فقط نمای انتزاعی به کاربر ارائه می‌شود، در حالی که داده‌های پیچیده و دقیق از کاربر پنهان می‌شوند. از سوی دیگر، در Encapsulation ، اعضای داده و توابع عضو به عنوان یک واحد واحد بسته‌بندی می‌شوند و می‌توانند با استفاده از اصلاح‌کننده‌های دسترسی و متدهای getter و setter محافظت یا در دسترس قرار گیرند

**Abstraction** در C# برای پنهان کردن داده‌های ناخواسته استفاده می‌شود و فقط ویژگی‌ها و روش‌های مورد نیاز را به کاربر نشان می‌دهد. Encapsulation در C# برای اتصال اعضای داده و توابع عضو به یک واحد واحد استفاده می‌شود تا از دسترسی مستقیم افراد خارجی به آن جلوگیری شود.



