|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Bellek Yönetimi ve Garbage Collector** | [**Favorilerime Ekle**](javascript:__doPostBack('ctl00$ctl00$ContentPlaceHolder1$ContentPlaceHolder1$linkFav','')) | | |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | [Sayfayı Yazdır](javascript:;) | [Sık Kullanılanlara Ekle](javascript:;) | [Arkadaşıma Gönder](javascript:;) | [MySpace](javascript:;) | [Del.Ico.Us](javascript:;) | [Digg](javascript:;) | [Facebook](javascript:;) | [Google](javascript:;) | [Mixx](javascript:;) | [Reddit](javascript:;) | [StumbleUpon](javascript:;) | | |
| Gün içerisinde, üzerinde çalıştığım uygulamalarda yada incelediğim kod parçalarında sıkça karşıma çıkan bir durum var. Kullanılmayan yada artık kullanılmayacak olan, atıl duruma düşen nesneler ya oldukları gibi bırakılıyor yada çok azına yapılması gereken bellek boşaltma işlemi uygulanıyor. Uzman olanımızdan tutun da aramıza yeni katılmış olan arkadaşlara kadar bir çok kişi de ortak bir fikir var, “garbage collector bizim yerimize halleder, bizim etliye-sütlüye karışmamıza gerek yok”. Yazılımcı olarak karşı çıktığım bir yaklaşım. Bize empoze edilen “iyi programcı tembel programcı” diktesinin payı olabilir mi bu işte? Ama burada bahsi geçen tembel kişi, yapması gereken işi en kısa yoldan yapan kişidir, yapması gereken işi yapmayan kişi değil. Microsoft platformlarını öne çıkartan nedenlerden biri de, kullanıcıların gereksinimlerini çok iyi ölçüyor olmasıdır. Visual Studio ve .NET framework’ü bunun en güzel kanıtı. Gereksinimlerimizin büyük bir kısmına yanıt veren bu uygulama geliştirme ortamları, kodlarımıza olan güvenimizi olması gerekenin biraz üzerine taşıyor sanırım. Peki bu konuda uygulamamız gereken doğru yaklaşım nedir? Doğru yaklaşım, garbage collector’ın işini kolaylaştırmak ve vereceği kararlarda ona yardımcı olmaktır. İsterseniz gelin hep birlikte konunun biraz daha derinine inelim.  Önce ki satırlarda bahsettiğim GC’ye yardımcı olma konusunda ne yapabiliriz? Konu hakkında MSDN’e bir göz atmakta fayda var. MSDN’e baktığımızda, bildiklerimizden çok ta farklı şeyler söylemediğini göreceğiz. Diyor ki;  Çöp toplayıcısı (garbage collector) heap’te herhangi bir nesne varsa halen bir uygulama tarafından kullanılıp kullanılmadığımı kontrol eder. Heap’te nesne varsa ve bu nesneler memory kullanıyorlarsa, kullandıkları bu memory’yi garbage collector tekrar kullanmak için geri isteyebilir (mecburi kaldığı durumlarda, memory taşması gibi). Eğer heap için kullanılabilecek daha fazla memory alanı yoksa (yani bir önceki adım da garbage collector yeterli memory alanı sağlayamadıysa) “**new**” operatörü ***OutOfMemoryException*** istisnası fırlatacaktır (işte burasa bir yazılımcı olarak kendimizi sorgulamamız gereken nokta).  İşin biraz daha derinine indiğimizde her uygulamanın root’a set edildiğini görürüz. Root’lar uygulamalarda kullanılan değişkenlerin kimliklerinin tanımlandığı, tutulduğu yerlerdir. Root’lar JIT (just- in-time) compiler’ı ve CLR (common language runtime) tarafından yönetilirler ve garbage collector algoritması tarafından erişilebilir kılınırlar.  Garbage collector çalışmaya başladığında tüm objelerin heap içersinde birer çöp olduğunu farz eder. Bir başka değişle uygulama (application) root’larının heap içersinde ki hiç bir nesneyi referas etmediklerini farzeder. Daha sonra çöp toplayıcı root üzerinde çalışmaya başlar ve root üzerinden erişilebilen tüm objelerin bir grafiğini çıkartır. Bu grafiğin “Şekil 1” de ki gibi olduğunu düşünerek bunun üzerinden konuşalım.  Sekil1Çöp toplayıcı heap üzerinde gezinirken sırasıyla A,B,C ve D nesneleri (nesnelerinin taşıdığı stack değerlerinin yerini temsil eden referanslar) ile karşılaşmış ve bunların grafiğini çıkartmıştır. Neden E nesnesinden bahsetmedim? Çünkü O’nu en sona sakladım. Dikkat ederseniz B ve E nesneleri şekil üzerinde birbirlerine bağlılar. Bunun anlamı, B nesnesinin E nesnesini temsil eden bir referans taşıdığıdır. Yani bir başka değişle B nesnesi, E nesnesinin depolandığı heap alanını gösteren bir referans taşıyor. Garbage collector referanslar içerisinde recursive şekilde çalışır. Bu sebepten dolayı okuma sırası E nesnesine gelmeden B nesnin referansı sayesinde E nesnesinden haberdar olur. B nesnesine E nesnesinin referansı şu şekilde atanmış olabilir;   |  | | --- | | **String** referans1 = "Referans 1"  **String** referans2 = **referans1**; |   Görüldüğü gibi bir referans tipi başka bir referans tipini referans etmiş :) . Garbage collector “Şekil 1” de bahsi geçen root üzerinde işini bitidiğinde, bir sonra ki root’a (her uygulamanın bir root’u olduğunu hatırlayalım) geçer ve aynı işlemleri orada da yapar. Bu işlem tüm root’ların kontrolü bitene kadar devam eder. Burada şunu belirtmek isterim; Garbage collector uygulamalar tarafından erişilen heap bellek bölgesinin bir haritasını çıkartırken buraların dolu mu yada boş mu olduğuna bakmaz.Bir uygulama tarafından bu bölgeye erişilip erişilmediğine bakar. “Şekil 1” üzerinde ki A ve C nesnelerinin null referanslar taşıdıklarını farzedecek olursak, yine de garbage collector’ın heap haritasına dahil edileceklerdir. Bu durum çöp toplayıcıya, A ve C nesnelerinin işgal ettikleri heap bölgelerinin artık temizlenme zamanının geldiğini anlatır.Sekil2-3  Buraya kadar güzel geldik. Bakalım çöp toplayıcı belleği nasıl boşaltacak ve bize top oynamak için daha geniş bir sahayı nasıl sağlayacak (bu aralar PES takılıyorum).  Çöp toplayıcı tüm application’lara ait root’ların haritasını çıkarttığında, elinde ki haritaya bakar. Uzun zamandır uygulamalar tarafından erişilmemiş olan yada uygulama tarafından boşaltılmış olan alanları heap’ten siler. **Bu silme işlemi sonunda boş kalan yerlere, çöp olmayan nesne referaslarını taşıyarak referans erişimlerinin daha hızlı olmasını sağlar.** Yine “Şekil 1” üzerinden gidecek olursak, A ve C nesnelerinin null referans taşıdıklarını, D nesnesine uzun zamandır uygulama tarafından erişim yapılmadığını, B ve E nesnelerinin ise halen bir application tarafından kullanıldıklarını düşünelim. Bu tablo üzerinde çalışacak olan çöp toplayıcı “Şekil 2” de ki gibi temizleme, “Şekil 3” de ki gibi kaydırma işlemi yapacaktır.  Buraya kadar yazılanlar genel olarak garbage collector’ın kullandığı algoritma ile ilgiliydi. Peki çöp toplayıcısının algoritmasına biz yazılımcıların nasıl yardımcı olabileceği noktasına geri dönersek karşımıza ne çıkar?  Bu aşamaya geldiğimizde **dispose metodları ,finalize işlemleri ve null değer atamaları** ile karşılaşırız. Dispose metodları ve finalize işlemleri, iyi bir bellek yönetimi için birbirleri ile bağlantılı bir şekilde kullanılırlar. *Dispose* metodları ile managed resources ve unmanaged resources üzerinde nasıl bir serbest bırakma işlemi uygulanacağına biz karar verebiliyorken, *finalization* kısmında yıkıcı (destructor) metodlar yardımıyla yıkım işleminden önce ne yapılması gerektiğini belirleyebiliyoruz. Eğer istersek garbage collector tarafından nesneye ait yıkıcı metodun atlanmasını yada bir şart dahilinde gerçekleşmesini de sağlayabiliyoruz. İsterseniz sonlandırma (finalization) olayını biraz açalım.  Finalization’ın ayrıntısına bakacak olursak “**finalization queue**” ve “**freachable queue**” adında iki yapı karşımıza çıkar. Her iki yapı da çöp toplayıcı tarafından kontrol edilen bir tür içsel veri yapısıdır. Yani memory’de ki veriler hakkında bilgi tutan yapılardır. Yarattığımız nesnenin yıkıcı (destructor) metodu varsa “new” operatörü kullanıldığında finalization queue’ye bu nesne için bir pointer eklenir. Burada dikkat etmemiz gereken bir nokta var. Destructor’ı olan bir nesne, “new” keyword’ü ile oluşturulduğu anda finalization queue’ye ekleniyor. GC’nin bunun için bir çaba harcamasına gerek olmadığı gibi burada sadece çöp nesnelerin bilgileri yok. Bir destrunctor’a sahip olan tüm nesnelerin pointer’ları var. Garbage collector*finalization queue sayesinde hangi nesnelerin destructor metodlarını olduğunu bilir*.  Garbage collector çalışmaya başladığında, daha önceden oluşturmuş olduğu root grafiğine bakarak, burada bulunan çöp halinde ki nesnelerin finalization kuyruğunda pointer’larının olup olmadığını kontrol eder. Finalization kuyruğunda pointer’ı olan çöp nesnelere ait pointer’ları, finalization queue’den siler ve freachable queue’ye ekler. Freachable kuyruğu artık **destructor** metodları çağrılmaya hazır nesnelerin pointer’larını barındırır. Dikkat ettiyseniz destructor metoda sahip çöp olmayan nesnelerin pointer’ları finalization queue’de durmaya devam ediyor.  Özel bir runtime thread’i ise freachable kuyruğu üzerinde ki hareketleri takip ederek bu kuyrukta ki nesnelere ait destructor metodlarını çağırarak kuyruğu boşaltır.  Finalization işlemlerini inceledikten sonra dispose metodlarına bakacak olursak şu şekilde giriş yapmak anlaşılır olur; dispose metodları, yönetilen ve yönetilmeyen kaynakların finalize işlemini beklemeden serbest bırakmamıza olanak sağlar. Managed ve Unmanaged kaynakları finalize sırasında da serbest bırakabiliriz ancak*finalize işleminin ne zaman tetikleneceği belli değildir*ve bu işlem tetiklenene kadar kullandığımız nesneler boşuna kaynak işgal ederler.  Dispose metodları içerisinde, garbage collector tarafından nesnemizin finalize işlemine maruz bırakılmamasını sağlayabilir ya da belli şart yada şartlar oluşana kadar finalize işleminin uygulanmasını da engelleyebiliriz.   |  |  | | --- | --- | | Not | Kaynak işgal etmek denilince sadece memory üzerinde yapılan alan işgali aklınıza gelmesin. Açık bırakılan bir connection yada kapatılmayan bir dosya yazma işlemi de kaynak işgalidir. Açık bırakılan connection server’ızda bağlantı işgali, açık bırakılan yazma işlemi ise dosya işgali yapar. |   Tüm bunları bilmeden iyi bir bellek yönetimi yapmak mümkün değildir. İlk önce elimizde ki araçları iyi tanımalı ve bu araçların yetenekleri doğrultusunda kendimize yöntemler belirlemeliyiz. Teoriyi kuvvetlendirmek adına, dispose metodlarını kullanmak için IDispose arayüzünü uygulayan ve bir yıkıcı metoda sahip olan aşağıda ki nesnemiz üzerinden konuşalım.   |  | | --- | | **using** System;  **namespace** Uygulama {     **public class** MyClass : **IDisposable**     {         **public void** Dispose()         {             // Nesnemiz içersinde kullanmış olduğumuz             // yardımcı nesnelerin kaynakları burada serbest bırakılır.         }          ~MyClass()         {              // Finalize gerçekleşmeden önce yapmamız             // gereken işlemleri burada gerçekleştiririz.         }      } } |   Örnek nesnemize IDisposable arayüzü ve destructor metodu uygulandı. Uygulamamız içersinde “MyClass” isimli nesnemiz ile işimiz bitip Dispose() metodunu çağırdığımızda, ilgili metod içersinde yazdıklarımız çalışacaktır. Burada bilhassa unmanaged (yönetilmeyen) kaynakları işlemeliyiz. Belllek taşması yada başka hehangi bir sebepten dolayı nesne direk olarak finalize işlemine tabi kaldığında yıkıcı metodumuz olan ~MyClass() devreye girecek, burada yazdıklarımızı çalıştıracak ve ek olarak managed (yönetilen) kaynakları işleyecektir.   |  |  | | --- | --- | | Not | Eğer yazmış olduğumuz nesneyi başka bir nesneden türetmişsek, kendi nesnemiz içerisinde ki Dispose() metodunu oluştururken base class’ımıza ait dispose metodunu da mutlaka çağırmalıyız (eğer IDisposable arayüzünü uygulayan bir nesne ise) . |   Bellek yönetiminin finalize kısmını tamamen biz el almak istediğimizde garbage collector’a ait iki önemli metod karşımıza çıkar. **GC.SuppessFinalize** ve **GC.ReRegisterForFinalize** metodları.  **GC.SuppessFinalize** metodunu, kullanmış olduğumuz objelerin garbage collector tarafından sonlandırma işlemine (finalize) tabi tutulmasını engellemek için kullanılırız. MSDN tarafından SuppressFinalize metodunu kullanmak için önerilen bir desen mevcuttur. Desen üzerinde düşünüldüğünde gayet mantıklı ve kullanışlıdır. Nedir MSDN tarafından bize önerilen desen;   |  | | --- | | **class** MyClass : **IDisposable**     {         /// <summary>         /// Nesnemize ait yıkıcı (destructor) metodumuz.         /// </summary>         ~MyClass()         {             // Bir şekilde destroctor tetiklendiğinde yönetilmeyen nesneleri ele              // almalıyız. Bunu da Dispose() metodumuzu burada çağırarak              // sadece unmanaged resources'ın ele alınması sağlıyoruz.             // İş buraya geldiğinde managed resources kısmını zaten çöp toplayıcının              // kendisi ele alacak.             **Dispose(false);**         }          /// <summary>         /// IDisposable arayüzü uygulandıktan sonra içeriğini bizim doldurmamız          /// gereken Dispose metodu.         /// </summary>         **public** **void** Dispose()         {             // Dispose metoduna "true" değeri verilerek hem yönetilen hem de              // yönetilemeyen kaynakların serbest bırakılmaları sağlanıyor.             Dispose(true);             // Önce ki adımda yönetilen ve yönetilmeyen kaynakların tamamı              // ele alındığı için destructor'a gidilmesine artık gerek yok.             **GC.SuppressFinalize(this);**         }          /// <summary>         /// IDisposable arayüzünün uygulanması sonucu doldurulması gereken          /// Dispose() metodunun görevini kısmen yerine getirecek ve parametre         /// olarak verilen değere göre hangi kaynaklar üzerinde işlem          /// yapacağımıza karar vereceğimiz metodumuz.         /// </summary>         /// <param name="disposing">         /// Managed yada Unmanaged kaynaklarından hangisinde çalışacağımızı          /// belirten parametre. False verildiğinde sadece unmanaged kaynaklar          /// üzerinde işlem yapılır.         /// </param>         **private void**Dispose(bool disposing)         {             if (disposing)             {                 /\*                  \* (Yönetilen Kaynaklar / Managed Resources)                  \* Nesnemize bağlı diğer nesnelerin Close/Dispose metodları çağrılmalıdır.                   \* Mesela nesnemiz bir SqlConnection sınıfını devralan bir nesne olabilir.                   \* Bu durumda, kodun bu kısmında SqlConnection nesnemizin Close()                   \* metodunu çağırarak açık olan bağlantıyı kapatabilir ve Dispose() metodunu                   \* çağırarak kullandığı kaynakları serbest bırakmasını sağlayabiliriz.                 \*/             }              /\*              \* (Yönetilmeyen Kaynaklar / Unmanaged Resources)              \* Nesnemiz içersinde, operasyonlara yardımcı olacak nesneler kullandıysak               \* bunların kullandıkları kaynakları serbest bırakmak yine bizim sorumluluğumuzda               \* olan görevdir. Kodun bu kısmında bu işlemi gerçekleştiririz.              \*/         }     } |   Yukarıda ki desenin içerisine yerleştirmiş olduğum açıklamalar aslında gayet bilgilendirici. Ancak bir özet yapmam gerekirse; IDisposable ararayüzünun uygulanması ile gelen Dispose() metodunu kısmen taklit edecek bir metod yazıyoruz ve yazdığımız bu metod içerisinde yönetilen ve yönetilmeyen kaynakları ele alıyoruz. Duruma göre de nesnenin destructor’ına girilmesinin (sonlandırma işlemine tabi kalmasının) önüne geçiyoruz. Yukarıda ki desen IDisposable.Dispose(), Dispose(bool disposing) ve ~MyClass() sırası ile okunmalı.   |  |  | | --- | --- | | Not | IDispose arayüzünü implemente ederken benim uymadığım bir yönerge var. Dispose metoduna birden fazla kez çağrı yapılırsa, yapılan bu çağrıların ilkine yanıt vermesi, diğerlerine ise ObjectDisposeException istinası fırlatması gerekir. Bu işin nasıl yapılacağını size bırakıyorum. Ne yapmanız gerektiği kafanızda canlanmadıysa MSDN’de IDisposable arayüzü ile ilgili ufak bir tarama ile doğru bilgiye ulaşabilirsiniz. |   Aşağıda ki örneğimizde nasıl bir modelleme yaptığımı açıklamam gerekirse; İlk önce IDisposable arayüzünü uygulayan, 2 yapıcı metodu, Close() ve Open() metodları olan “*MySqlConnection*” isimli bir nesne oluşturdum. Daha sonra bu nesne içerisinde ki unmanaged resources olayını gösterebilmek için “*MyConnectionString*” isimli bir nesne oluşturarak, “MySqlConnection” nesnem içerisinde kullandım.   |  | | --- | | **using** System; **using** System.Data.SqlClient;  **namespace** DatabaseConnection {     **class** MyConnnectionString     {         **public** MyConnnectionString(String connectionString)         {             **this**.ConnectionString = connectionString;         }  **public** String ConnectionString         {             get;             set;         }     }  **class** MySqlConnection : **IDisposable**     {         **private** SqlConnection m\_connection;         **private** MyConnnectionString m\_connectionString = **new** MyConnnectionString("bağlantı bilgisi");  **public** MySqlConnection()         {             m\_connection = **new** SqlConnection(m\_connectionString.ConnectionString);         }  **public** MySqlConnection(SqlConnection connection, String connectionString)         {             m\_connection = connection;         }  **public** **bool** Open()         {             bool baglantiBasarisi = true;             try             {                 m\_connection.Open();             }             catch             {                 baglantiBasarisi = false;             }  **return** baglantiBasarisi;         }  **public** **bool** Close()         {             bool kapamaBasarisi = true;             try             {                 if (m\_connection.State != System.Data.ConnectionState.Closed) m\_connection.Close();             }             catch             {                 kapamaBasarisi = false;             }              return kapamaBasarisi;         }  **public void Dispose()**         {             **Dispose(true);**             **GC.SuppressFinalize(this);**         }  **private void Dispose(bool disposing)**         {             if (disposing)             {                 m\_connection.Close();                 m\_connection.Dispose();             }              m\_connectionString = null;         }          ~MySqlConnection()         {             **Dispose(false);**         }      } } |   Vermiş olduğum örnekte görüldüğü gibi “SuppressFinalize” metodu gerçekten de çok kullanışlı. Bu metod sayesinde nesnemizle işimiz bittikten sonra zaten bizim bizzat serbest bıraktığımız kaynaklarla destructor’ın yeniden uğraşmasını engelliyoruz. Bu da bize performans olarak geri dönüyor. Garbage collector’ın sistem kaynaklarını mümkün olan en kısa zamanda serbest bırakmasına yardımcı olmuş oluyoruz. Bir sebepten dolayı nesne Dispose() metodunu çağrmadan finalize işlemine girecek olursa, yönetimini Dispose() olayına verdiğimiz unmanaged kaynakları ele alıyoruz. Bu kısım çok önemli. Çünkü bizim elimizde olmayan bir sebepten dolayı bellek yetersiz kalır ve garbage collector bellek alanı açmak için nesnemizin kullandığı bellek alanını serbest bırakmak isterse, direk olarak nesnemize ait yıkıcı metod çağrılacaktır.   |  |  | | --- | --- | | Not | Oluşturduğumuz nesnemizin implemente ettiği arayüzlerden birinde yada devraldığı obje de “**Close**” metodu varsa, *dispose*metodunu oluştururken bu *Close()* metodunu da mutlaka çağırmalıyız. |   **GC.ReRegisterForFinalize** metodunu ise nesnemizin belli şart yada şartlar sağlanıncaya kadar yok edilmesini istemiyorsak kullanırız. GC.SuppessFinalize metodunu anlatırken kullandığımız örneğimizi bu durum için uyarlayalım. Bunun için nesnemizin yıkıcı metodu üzerinde çalışmamız yeterli olacak.   |  | | --- | | ~MySqlConnection() {      // Eğer connection nesnemiz bir sql komutunu halen çalıştırır durumdaysa      // yokedilmesini engelliyoruz ve finalization kuyruğuna (finalization queue)      // yeniden ekliyoruz.     if (m\_connection != null &&**m\_connection.State == System.Data.ConnectionState.Executing**)      {         **GC.ReRegisterForFinalize(this);**      }      else      {         **Dispose(false);**      } } |   Örnekte şunu yaptık; Eğer “m\_connection” üyesi halen bir sql komutu üzerinde çalışıyorsa yokedilmesini engelleyerek finalization queue’ye yeniden eklenmesini sağladık. Yeniden ekledik çünkü nesnenin yıkıcı metodu çağrıldığında finalization kuyruğundan silinmiş oldu. Ancak biz nesnemizin sonlandırma işlemine maruz kalmamasını değil, uygun koşul sağlandığı zaman nesnemizin yok edilmesini istiyoruz. Bunun için nesnemizin yıkıcı metodu üzerinde çalışmamız yeterli olacak.  Garbage collector’ın algoritmasını ve bizim bu algoritmayı en verimli şekilde nasıl kullanabileceğimiz gördük. O halde aşadı ki örnek kod parçası üzerinde biraz düşünelim. Sizce bu örnek kodda ki yanlışlar ve doğrular neler?   |  | | --- | | **int** i = 1000; ArrayList arrayList = new ArrayList(1000);  **for** (**int** count = 0; count < i; count++) {      MySqlConnection connect = **new** MySqlConnection();      arrayList.Add(connect); } |   Garbage collector’ın algoritmasını ve bizim bu algoritmayı en verimli şekilde nasıl kullanabileceğimiz gördük. O halde aşadı ki örnek kod parçası üzerinde biraLütfen ilerleyen satırlarda ki örneğe bakmadan biraz daha düşünün. Eğer cevaplarınız hazır ise aşağıda ki örnek kod parçasına bakın. düşünelim. Sizce bu örnek kodda ki yanlışlar ve doğrular neler?   |  | | --- | | ArrayList arrayList = new ArrayList(1000);  **for** (**int** count = 0; count < i; count++) {      MySqlConnection connect = **new** MySqlConnection();      arrayList.Add(connect);      connect = **null**  } |   Sizce şimdi nasıl oldu? Bence fena sayılmaz. Burada yaptığımız şu; yarattığımız MySqlConnection türünden nesnelerimizi arrayList içersine ekliyoruz. Ancak elimizde bir nesneyi temsil eden iki tane referansımız olmuş oldu. Bir tanesi “arrayList” nesnesi içersinde diğeri ise “connect” nesnesinin heap bölgesinde. Peki bizim bu iki referansa ihtiyacımız var mı? Hayır, yok. Bizim sadece arrayList nesnesinin taşıdığı referansa ihtiyacımız var. O halde connect isimli nesnemize null atayarak çöp nesne haline getiriyoruz. Ayrıca bir önceki kod parçasında “int” türünden kullandığımız “i” değişkenini burada kullanmadık. Çünkü bunu memory’de tutmaya gerek yok. Bir değeri direk olarak kulanma imkanımız varsa bunun için bir nesne yaratarak bellek işgali yapmamıza gerek yok. Bu tür gereksiz memory kullanımından sakınmalıyız.  Peki dispose metodlarını nerede kullanacağız? Dispose metodlarının amacı nesnemiz içersinde kullanılan kaynakları serbest bırakmak olduğundan, nesnemiz ile işimiz biter bitmez, bu kaynakları serbest bırakmak için kullanacağız.   |  | | --- | | MySqlConnection connect = **new** MySqlConnection(); connect.Open(); /\* \* Bir takım işlemler... \*/ connect.Close();  // Nenemizin kullandığı kaynakları serbest bıraktık. connect.Dispose(); // Nesnemizin referansının heap üzerinde işgal // ettiği alanı boşaltarak nesneyi çöp yaptık. connect = **null;** |   Konuyu daha fazla uzatmadan bitirmek istiyorum. Tüm makalenin özeti aslında şu; İşiniz biten nesnenin ne yapılacağı konusunda karar verme işini tamamen garbage collector’a bırakmayın ve GC’nin bir kahraman olmadığını unutmayın.  Eğer siz nesne ile işinizin bittiğini çöp toplayıcısına bildirirseniz, çöp toplayıcı çöpleri toplamak için çalıştığında, nesnenizi ne kadar süredir kullanmadığınıza bakmayacaktır ve onun bir çöp nesne olduğunu bilerek davranacaktır. Ancak siz garbage collector çalışmadan kısa bir süre önce nesne(ler) ile işinizi bitirir ancak bunu garbage collector’e bildirmezseniz, GC bu(nların) çöp olduklarını anlamayacak ve kullandıkları bellek alanlarını serbest bırakmayacaktır. Bu nesne(ler)de bellekte yer işgal etmeye ve uygulamalarınızı yavaşlatmaya devam edecek(ler)dir. Bilhassa uygulamalarınız bir server üzerinden çok sayıda kullanıcıya hizmet veren uygulamalar ise (WCF kullanan windows desktop uygulamaları yada ASP.NET uygulamaları gibi) bellek yönetimine çok daha fazla dikkat etmeniz gerekir.  Kaynakları hor kullanmak bir fırın ustasının 10kg undan 3kg ekmek yapmasına benzer. Usta ekmekleri yaparken unu da sağa sola savurarak kullanıyorsa, o fırının sahibine acımak lazım. Bu usta gibi olmak istemiyor, gerçek anlamda bir usta (professional) olmak istiyorsak ilk önce elimizde ki kaynakları idareli kullanmalı, 10kg undan en az 8kg ekmek çıkarmalıyız. Eksik olan 2kg’da ekmeğin ıstakaya yapışmaması için kullandığımız un (uygulamamız tarafından hali hazırda aktif bir şekilde kullanılan bellek alanları) olmalı. Yani kaynaklarımızı yine gerektiği gibi kullanmış olduk.  Garbage collector’ınıza zeval gelmemesi dileği ile, |