فصل 7

کلاس ها و ارث بری

* ارث بری کلاس
* دسترسی به عضو های موروثی
* تمام کلاس ها از کلاس object مشتق شده اند
* پوشاندن عضو های کلاس پایه
* دسترسی به کلاس پایه
* استفاده از منابع یک کلاس پایه
* اجرای سازنده
* ارث بری در بین اسمبلی ها
* اصلاحگر های دسترسی عضو
* عضو های انتزاعی
* کلاس های انتزاعی
* کلاس های مهر و موم شده
* کلاس های استاتیک
* متدهای الحاقی
* قراردادهای نامگذاری

ارث بری کلاس

وراثت اجازه تعریف کلاس جدیدی را می دهد که شامل و گسترش یافته یک کلاس تعریف شده باشد.

* می توان از یک کلاس موجود (که کلاس پایه[[1]](#footnote-1) نامیده می شود) به عنوان پایه و اساسی برای یک کلاس جدید (که کلاس مشتق[[2]](#footnote-2) نامیده می شود) استفاده کرد.
  + عضو های کلاس خودش
  + عضو های کلاس پایه
* برای تعریف یک کلاس مشتق شده، مشخصات کلاس پایه را بعد از نام کلاس اضافه می کنیم. مشخصات کلاس پایه شامل شده است از دو نقطه و نام کلاسی که به عنوان کلاس پایه استفاده می شود. کلاسی مشتق شده است که به طور مستقیم از کلاس پایه ارث بری کند.
* یک کلاس مشتق شده را گسترش یافته کلاس پایه گویند، زیرا شامل عضوها و قابلیت های موجود در کلاس پایه است.
* کلاس مشتق شده می تواند هر عضو ارث برده شده را حذف کند.

به عنوان مثال، در زیر کلاسی را با نام OtherClass نشان می دهد، که از کلاس SomeClass ارث بری می کند:

|  |
| --- |
| خصوصیات کلاس پایه  ↓  class OtherClass : SomeClass  { ↑ ↑  ... دو نقطه کلاس پایه  } |

شکل 1-7 نمونه ای از هر یک از کلاس ها را نشان می دهد. کلاس SomeClass، در سمت چپ قرار دارد که دارای یک فیلد و یک متد است. کلاس OtherClass در سمت راست قرار دارد و از کلاس SomeClass مشتق شده است و یک فیلد و یک متد بیشتر از کلاس مشتق شده خود دارد.



شکل 1-7. کلاس پایه و کلاس مشتق شده

دسترسی به عضو های موروثی

عضو های موروثی مانند عضو های تعریف شده خود کلاس مشتق شده در دسترس هستند. (سازنده های موروثی کمی متفاوت تر هستند که در همین فصل آنها را بررسی خواهیم کرد.) به عنوان مثال، کد زیر کلاس های SomeClass و OtherClass که در شکل 7.1 نشان داده شده است را تعریف کرده است. این کد نشان می دهد که تمام چهار عضو کلاس OtherClass می توانند به صورت یکپارچه در دسترس باشند صرف نظر از اینکه آیا در کلاس پایه یا در کلاس مشتق تعریف شده اند.

* متد Main شی ای از کلاس OtherClass ایجاد کرده است.
* دو خط بعدی در متد Main، متد Method1 از کلاس پایه را با استفاده از فیلد های Field1 در کلاس پایه و Field2 در کلاس مشتق شده فراخوانی می کند.
* دو خط بعدی در متد Main، متد Method2 از کلاس مشتق شده را با استفاده از فیلد های Field1 در کلاس پایه و Field2 در کلاس مشتق شده فراخوانی می کند.

|  |
| --- |
| class SomeClass // کلاس پایه  {  public string Field1 = "base class field ";  public void Method1( string value ) {  Console.WriteLine("Base class -- Method1: {0}", value);  }  }  class OtherClass: SomeClass // کلاس مشتق شده  {  public string Field2 = "derived class field";  public void Method2( string value ) {  Console.WriteLine("Derived class -- Method2: {0}", value);  }  }  class Program  {  static void Main() {  OtherClass oc = new OtherClass();  oc.Method1( oc.Field1 ); // متدی از کلاس پایه با فیلدی از کلاس پایه  oc.Method1( oc.Field2 ); // متدی از کلاس پایه با فیلدی از کلاس مشتق شده  oc.Method2( oc.Field1 ); // متدی از کلاس مشتق شده با فیلدی از کلاس پایه  oc.Method2( oc.Field2 ); // متدی از کلاس مشتق شده با فیلدی از کلاس مشتق شده  }  } |

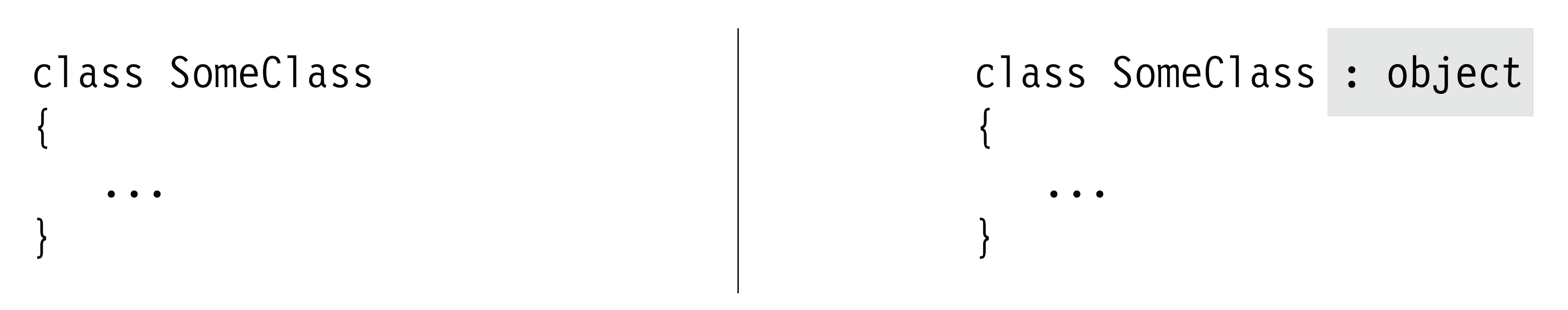
این کد خروجی زیر را تولید می کند:

|  |
| --- |
| Base class -- Method1: base class field  Base class -- Method1: derived class field  Derived class -- Method2: base class field  Derived class -- Method2: derived class field |

تمام کلاس ها از کلاس object مشتق شده اند

تمام کلاس ها، بجز کلاس object کلاس هایی مشتق شده هستند، حتی اگر آنها قسمت مشخصات کلاس پایه را نداشته باشند. کلاس object تنها کلاسی است که مشتق نشده است، زیرا پایه سلسله مراتب ارث بری است.

کلاس هایی که قسمت مشخصات کلاس پایه را ندارند، به طور ضمنی از کلاس object ارث بری می کنند. قرار ندادن مشخصات کلاس پایه مختصر نویسی برای تعیین کلاس پایه object است. همانطور که در شکل 2-7 نشان داده شده است، دو نوع معنایی معادل هستند.

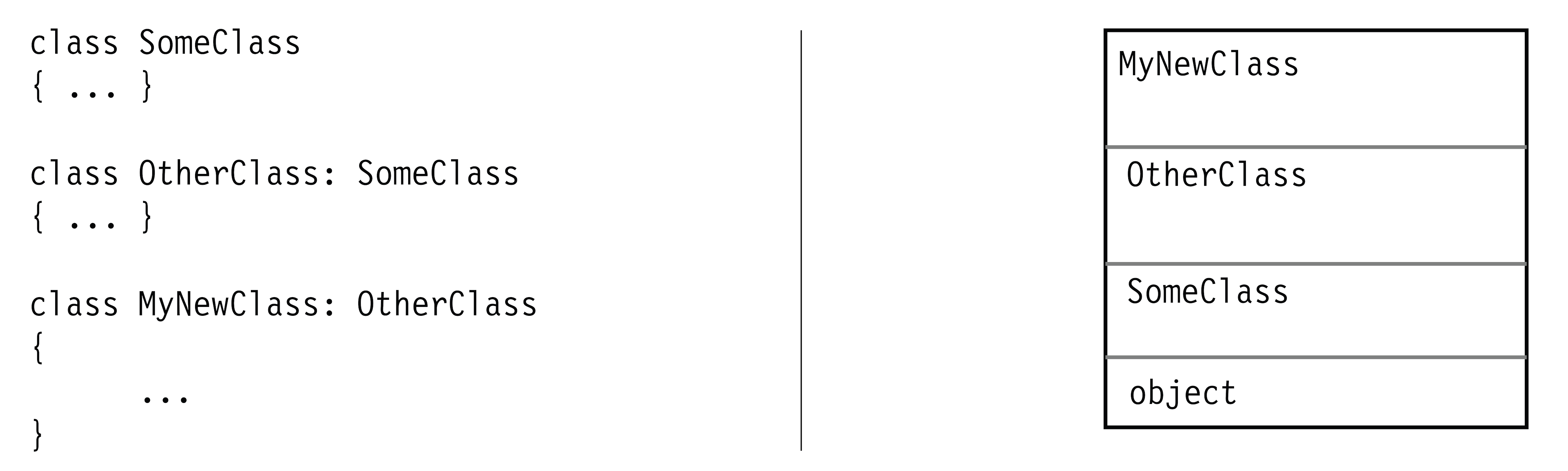


شکل 2-7. کلاس تعریف شده در سمت چپ به طور ضمنی و کلاس تعریف شده در سمت راست به صراحت از کلاس object ارث بری می کنند. و دو نوع معنایی معادل هستند.

مواردی که مهم است درباره ارث بری کلاس ها بدانید به شرح زیر است:

* در تعریف کلاس تنها می توان از یک کلاس برای مشخصات کلاس پایه استفاده کرد. (یک کلاس تنها می تواند از یک کلاس ارث بری کند.) که ارث بری منفرد[[3]](#footnote-3) نامیده می شود.
* اگر چه یک کلاس تنها می تواند از یک کلاس ارث بری کند اما هیچ محدودیتی در سطح ارث بری وجود ندارد. به این معنا که، کلاس تعیین شده به عنوان کلاس پایه ممکن است از کلاس دیگری ارث بری کند و به همین ترتیب تا زمانی که به کلاس object برسد ارث بری می کند.

کلاس پایه و کلاس مشتق شده شرایطی نسبی دارند. تمام کلاس ها، کلاس هایی مشتق شده هستند (یا از کلاس object یا از کلاس دیگری ارث بری می کنند) ، بنابراین به طور کلی زمانی که ما می گوییم یک کلاس، کلاسی مشتق شده است، منظور ما این است که مستقیماً از کلاس دیگری بجز کلاس object ارث بری می کند. شکل 3-7 سلسله مراتب ارث بری یک کلاس را نشان می دهد. به دلیل اینکه همه کلاس ها در نهایت به کلاس object ختم می شود آن را در تصاویر نشان نمی دهیم.



شکل 3-7. سلسله مراتب یک کلاس

پوشاندن عضو های کلاس پایه

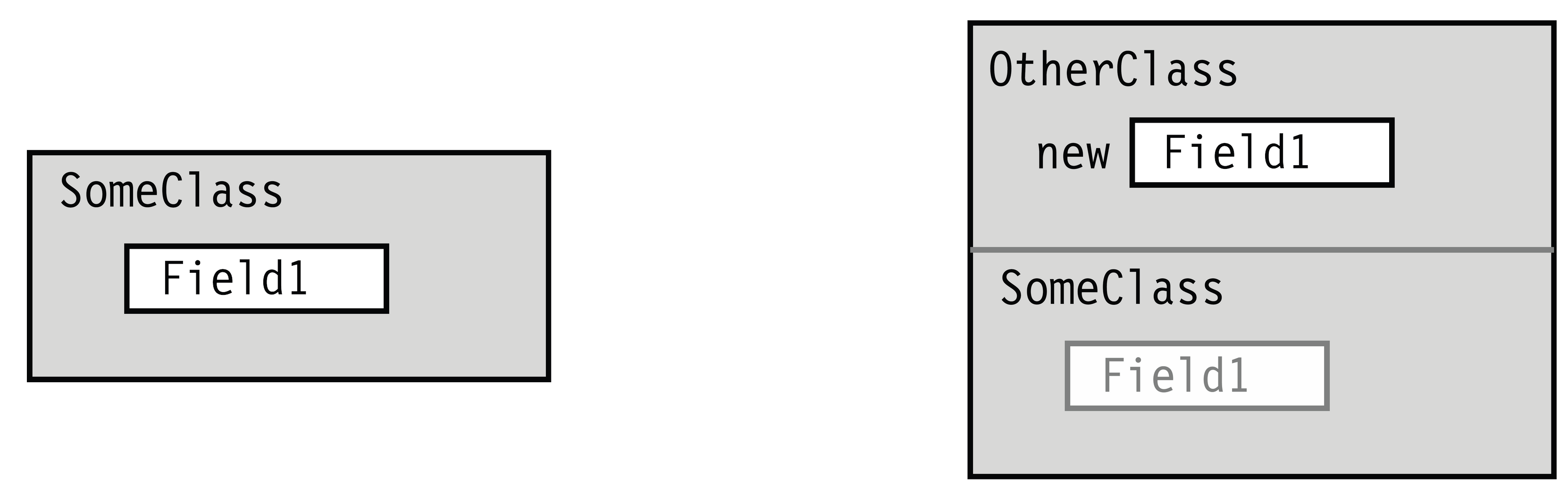
کلاس مشتق شده می تواند هر یک از عضو های موروثی را حذف کند، کلاس مشتق شده می تواند عضوی از کلاس پایه را با عضوی با همان نام پنهان کند. این کار بسیار مفید و یکی از ویژگی های عمده در ارث بری است.

به عنوان مثال، ممکن است بخواهید از یک کلاس پایه که متد خاصی را دارد ارث بری کنید. اگر چه، این متد برای کلاس کامل و بی نقص تعریف شده باشد، و ممکن است دقیقا همان چیزی را که در کلاس مشتق شده می خواهید انجام ندهد. در چنین حالتی آنچه شما برای انجام این کار می خواهید این کار است که متد کلاس پایه را با تعریف کردن یک عضو جدید در کلاس مشتق شده بپوشانید. برخی از جنبه های مهم پوشش یک عضو کلاس پایه در کلاس مشتق شده به شرح زیر است:

* برای پوشاندن داده های عضو موروثی، عضو جدیدی از همان نوع و با همان نام تعریف می شود.
* برای پوشاندن توابع عضو به ارث رسیده، عضو جدیدی با همان امضا تعریف می شود. به یاد داشته باشید که امضا شامل نام و لیست پارامتر ها است اما نوع بازگشتی را شامل نمی شود.
* برای اینکه کامپایلر بداند عمداً یک عضو موروثی پوشیده می شود از اصلاحگر new استفاده می شود. بدون آن، برنامه با موفقیت کامپایل خواهد شد، اما کامپایلر هشدار می دهد که عضو به ارث رسیده ای را پنهان می کند.
* همچنین می توان عضو های استاتیک را پنهان کرد.

کد زیر یک کلاس پایه و یک کلاس مشتق شده را تعریف می کند، و هر کدام دارای یک عضو با نام Field1 است. کلمه کلیدی new استفاده شده است تا به صراحت به کامپایلر بگوید در حال پوشاندن عضوی از کلاس پایه است. شکل 4-7 نمونه ای از هر دو کلاس را به تصویر کشیده است.

|  |
| --- |
| class SomeClass // کلاس پایه  {  public string Field1;  ...  }  class OtherClass : SomeClass // کلاس مشتق شده  {  new public string Field1; // پوشاندن عضو کلاس پایه با همان نام  ↑  کلمه کلیدی |



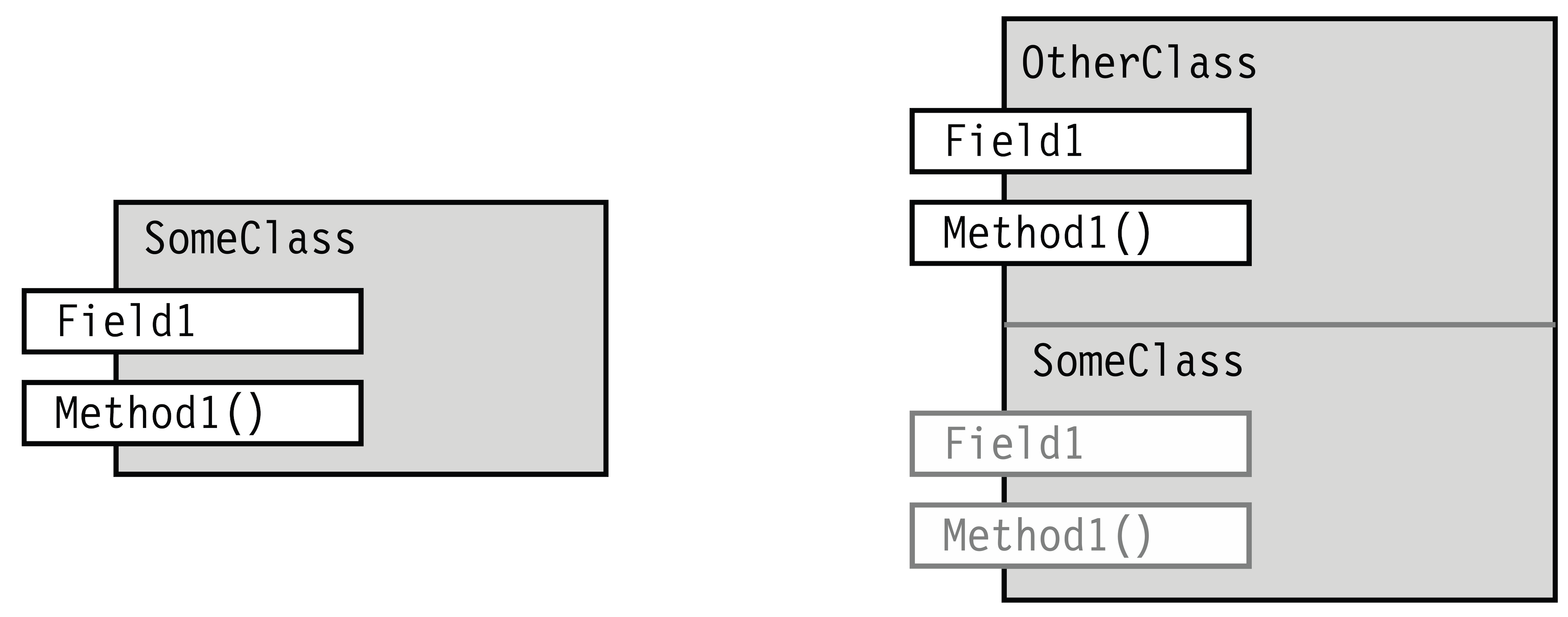
شکل 4-7. پوشاندن فیلدی از کلاس پایه

در کد زیر، کلاس OtherClass از کلاس SomeClass ارث بری می کند و هر دو عضو به ارث برده شده را پنهان می کند. توجه داشته باشید که از اصلاحگر new استفاده شده است. شکل 5-7 این کد را به تصویر کشیده است.

|  |
| --- |
| class SomeClass // کلاس پایه  {  public string Field1 = "SomeClass Field1";  public void Method1(string value)  { Console.WriteLine("SomeClass.Method1: {0}", value); }  }  class OtherClass : SomeClass // کلاس مشتق شده  { کلمه کلیدی  ↓  new public string Field1 = "OtherClass Field1"; // پوشاندن عضو کلاس پایه  new public void Method1(string value) // پوشاندن عضو کلاس پایه  ↑ { Console.WriteLine("OtherClass.Method1: {0}", value); }  } کلمه کلیدی  class Program  {  static void Main()  {  OtherClass oc = new OtherClass(); // ایجاد شی ای از کلاس مشتق شده  oc.Method1(oc.Field1); // استفاده از عضو های پوشش یافته  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| OtherClass.Method1: OtherClass Field1 |



شکل 5-7. پنهان کردن فیلد و متدی از کلاس پایه

دسترسی به کلاس پایه

اگر در کلاس مشتق شده ای باید به عضو پنهان شده ای دسترسی داشته باشیم، می توان با استفاده از دستور دسترسی به پایه[[4]](#footnote-4) به آن دسترسی داشت. همانطور که در زیر نشان داده شده است، این دستور شامل کلمه کلیدی base، نقطه و نام عضو است:

|  |
| --- |
| Console.WriteLine("{0}", base.Field1);  ↑  دسترسی به پایه |

به عنوان مثال، در کد زیر، کلاس مشتق شده ، فیلد Field1 در کلاس پایه را پنهان می کند اما با استفاده از دستور دسترسی به پایه می تواند به آن دسترسی پیدا کند.

|  |
| --- |
| class SomeClass { // کلاس پایه  public string Field1 = "Field1 -- In the base class";  }  class OtherClass : SomeClass { // کلاس مشتق شده  new public string Field1 = "Field1 -- In the derived class";  ↑ ↑  پنهان کردن فیلد کلاس پایه  public void PrintField1()  {  Console.WriteLine(Field1); // دسترسی به کلاس مشتق شده  Console.WriteLine(base.Field1); // دسترسی به کلاس پایه  } ↑  } دسترسی به پایه  class Program {  static void Main()  {  OtherClass oc = new OtherClass();  oc.PrintField1();  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| Field1 -- In the derived class  Field1 -- In the base class |

استفاده از منابع کلاس پایه

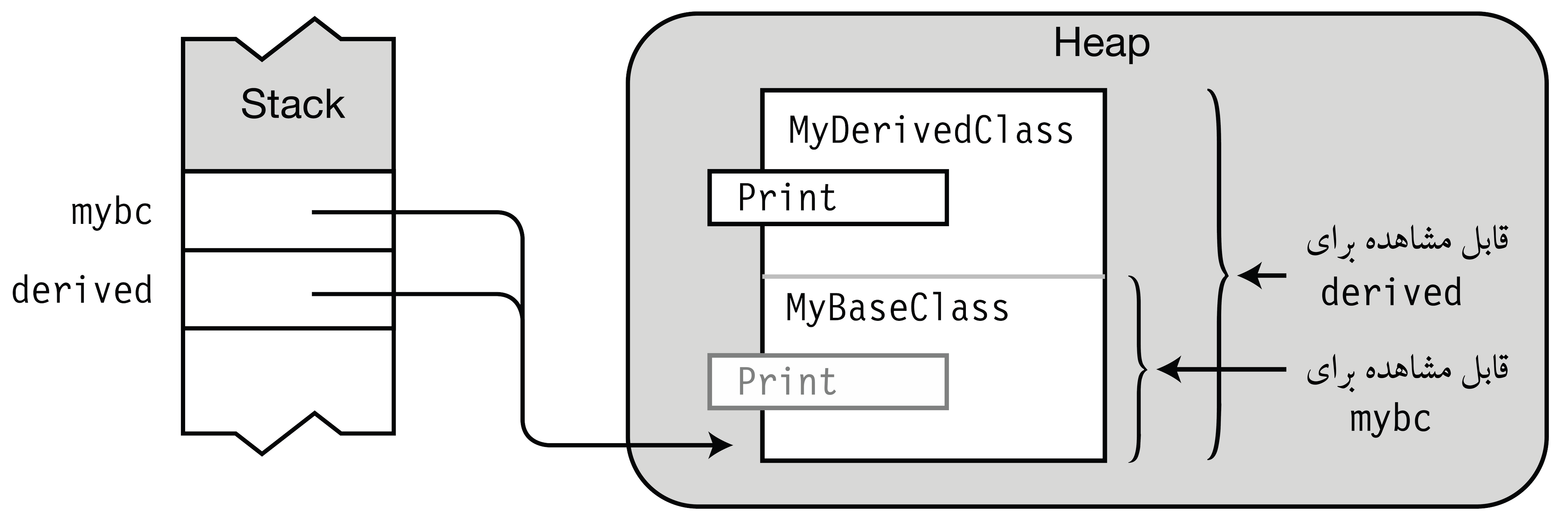
نمونه ای از یک کلاس مشتق شده، از نمونه ای از کلاس پایه و عضو های کلاس مشتق شده تشکیل شده است. ارجاع به این کلاس مشتق شده به کل شی کلاس، و همچنین به بخشی از کلاس پایه اشاره می کند.

اگر مرجعی از یک شی کلاس مشتق شده داشته باشید. می توانید با استفاده از عمل کست کردن[[5]](#footnote-5) (مرجع گیری)، مرجع ای برای بخشی از کلاس پایه تهیه کنید. عمل کست (مرجع گیری)، قبل از مرجع شی قرار می گیرد و از نام کلاس پایه که درون پرانتز قرار دارد تشکیل شده است. مرجع گیری با جزئیات بیشتری در فصل 16 بررسی خواهد شد.

در چند بخش بعدی، دسترسی به یک شی را با استفاده از ارجاع به بخشی از کلاس پایه بررسی می کنیم. می خواهیم با کد زیر شروع کنیم که مرجع ای برای شی تعریف می کند. شکل 6-7 کد را به تصویر می کشد و قسمتی از شی که توسط متغیرهای مختلف دیده می شود را نشان می دهد.

* خط اول، متغییر derived را ایجاد و مقداردهی می کند، که شامل مرجعی برای شی ای از نوع MyDerivedClass است.
* خط دوم، متغیری از نوع کلاس پایه MyBaseClass ایجاد می کند، و به کست متغییر derived ارجاع داده شده است. که مرجعی برای بخشی از کلاس پایه است.
  + مرجع بخشی از کلاس پایه در متغییر mybc ذخیره شده است.
  + مرجع بخشی از کلاس پایه نمی تواند باقی شی کلاس مشتق شده را ببیند، به دلیل اینکه از طریق یک مرجع به نوع پایه آن نگاه می کند.

|  |
| --- |
| MyDerivedClass derived = new MyDerivedClass(); // ایجاد شی  MyBaseClass mybc = (MyBaseClass) derived; // کست از مرجع |



شکل 6-7. مرجع derived می تواند کل شی MyDerivedClass را ببیند، در حالی که mybc فقط می تواند بخش شی MyBaseClass را ببیند.

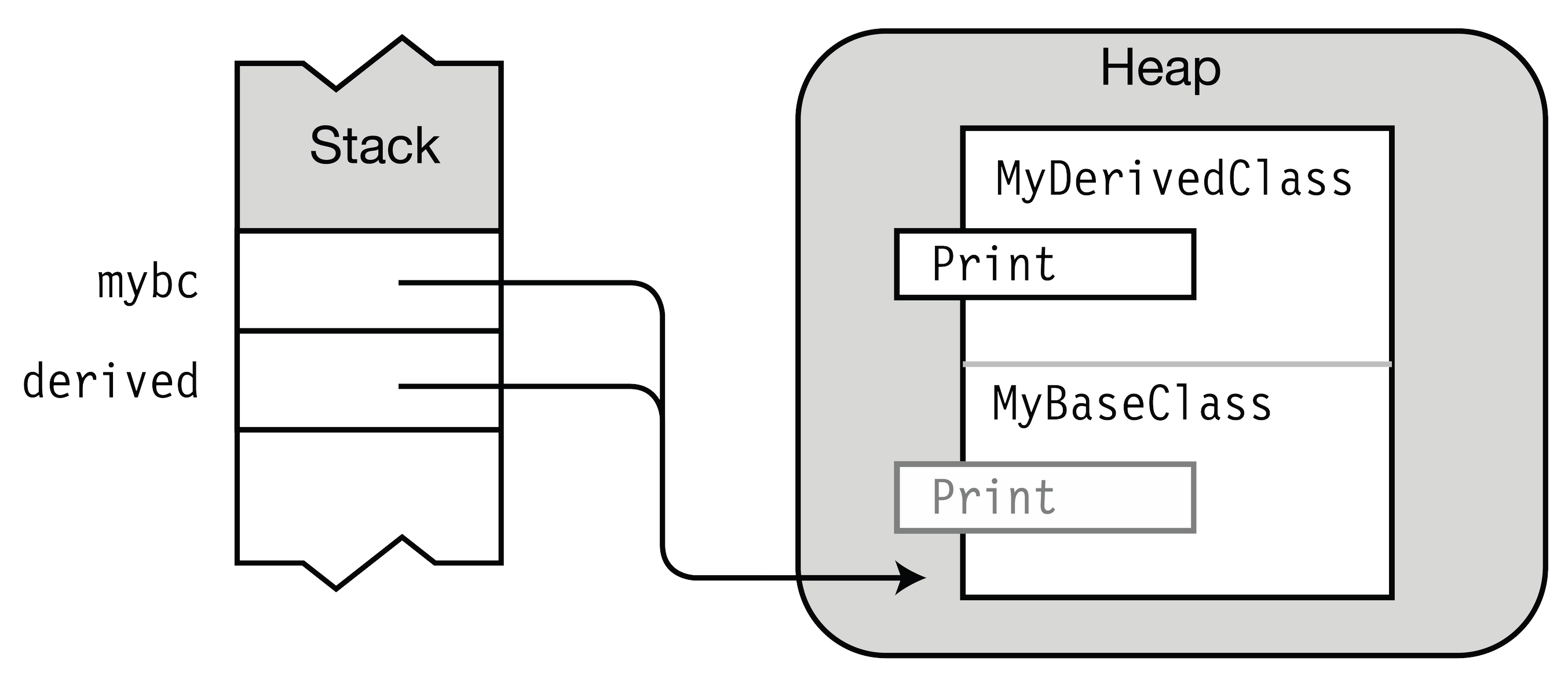
کد زیر تعریف و استفاده از این دو کلاس را نشان می دهد. شکل 7-7 شی و مرجع آن را به تصویر می کشد.

متد Main یک شی از نوع MyDerivedClass ایجاد می کند و مرجع آن را در متغییر derived ذخیره می کند. همچنین متد Main متغیری از نوع MyBaseClass نیز ایجاد می کند و از آن برای ذخیره مرجع کلاس پایه استفاده می کند. زمانی که متد Print از هر مرجع ای فراخوانی می شود، مرجع متدی را فراخوانی می کند که می تواند آن را ببیند، و متد Print برای هر مرجع رشته های مختلفی را در خروجی تولید می کند.

|  |
| --- |
| class MyBaseClass  {  public void Print()  {  Console.WriteLine("This is the base class.");  }  }  class MyDerivedClass : MyBaseClass  {  new public void Print()  {  Console.WriteLine("This is the derived class.");  }  }  class Program  {  static void Main()  {  MyDerivedClass derived = new MyDerivedClass();  MyBaseClass mybc = (MyBaseClass)derived;  ↑  مرجع گیری برای کلاس پایه  derived.Print(); // از طریق مرجع کلاس مشتق شدهPrintفراخوانی  mybc.Print(); // از طریق مرجع کلاس پایهPrint فراخوانی  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| This is the derived class.  This is the base class. |



شکل 7-7 مرجع ای برای کلاس مشتق شده و کلاس پایه

متد های Virtual و Override

در بخش قبل، مشاهده کردید که، زمانی که به کلاس پایه ای از یک شی دسترسی پیدا می کنید، تنها می توانید از عضو های این کلاس پایه استفاده کنید. متدهای مجازی به مرجع کلاس های پایه اجازه می دهند تا به کلاس مشتق شده دسترسی داشته باشند.

اگر موارد زیر رعایت شود، می توان با استفاده از مرجع کلاس پایه، متدی را در کلاس مشتق شده فراخوانی کرد:

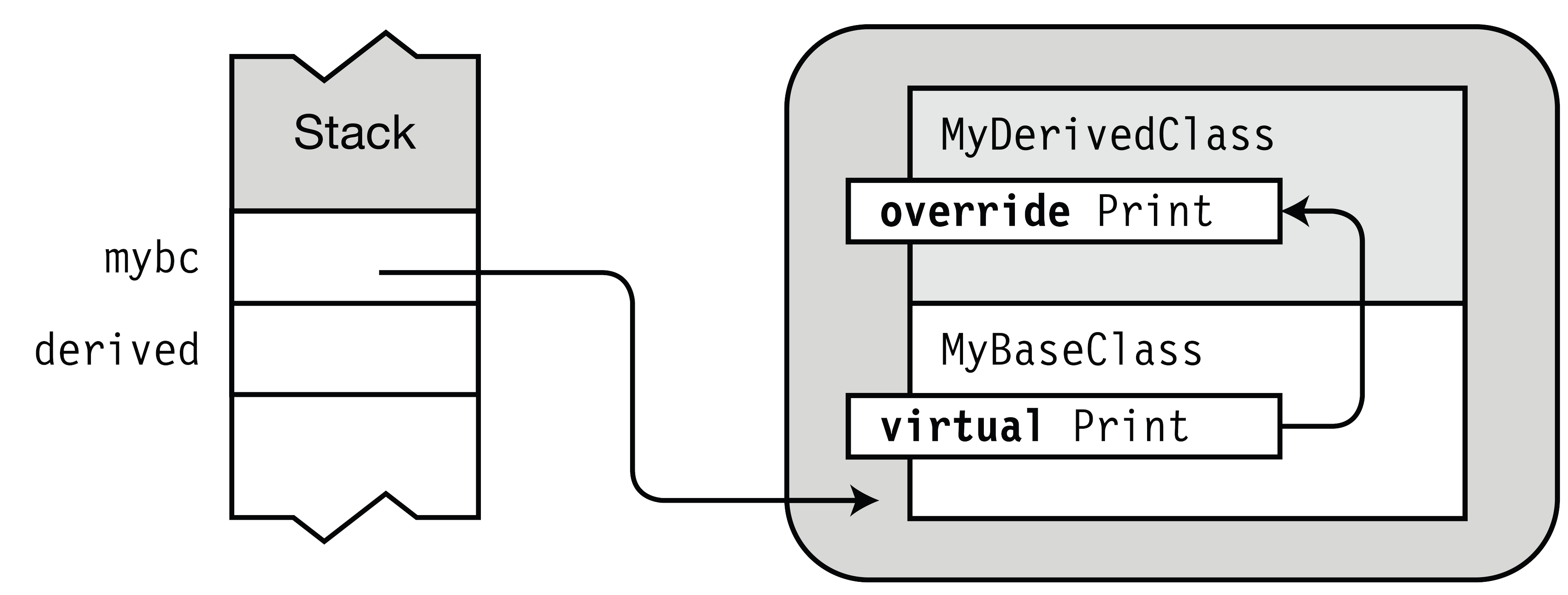
* متد کلاس مشتق شده و متد کلاس پایه دارای امضا و نوع بازگشتی یکسانی باشند.
* متد کلاس پایه با کلمه virtual برچسب گذاری شده باشد.
* متد کلاس مشتق شده با کلمه override برچسب گذاری شده باشد.

به عنوان مثال، کد زیر استفاده از اصلاحگر های virtual و override را در متدهای کلاس پایه و کلاس مشتق شده نشان می دهد.

|  |
| --- |
| class MyBaseClass // کلاس پایه  {  virtual public void Print()  ↑  ...  class MyDerivedClass : MyBaseClass // کلاس مشتق شده  {  override public void Print()  ↑ |

شکل 8-7 مجموعه ای از متد های virtual و override را به تصویر می کشد. به رفتار متفاوتی که نسبت به حالت قبل که از کلمه کلیدی new برای مخفی کردن متد ها کلاس پایه استفاده می کردیم، توجه داشته باشید.

* زمانی که متد Print با استفاده از مرجع کلاس پایه (mybc) فراخوانی می شود، فراخوانی این متد به کلاس مشتق شده منتقل می شود و متد کلاس مشتق شده اجرا می شود، زیرا:
  + این متد در کلاس پایه با اصلاحگر virtual تعریف شده است.
  + مطابق این متد، متدی در کلاس مشتق شده با اصلاحگر override وجود دارد.
* شکل 8-7 این کار با فلشی از متد virtual Print به متد override Print به تصویر کشیده شده است.



شکل 8-7. متد virtual و متد override

کد زیر همان کد بخش قبل است اما این بار متد ها با کلمات virtual و override برچسب گذاری شده اند. این کار نتیجه متفاوتی نسبت به مثال قبلی تولید می کند. در این مثال فراخوانی متد کلاس پایه باعث اجرا شدن متد کلاس مشتق شده می شود.

|  |
| --- |
| class MyBaseClass  {  virtual public void Print()  {  Console.WriteLine("This is the base class.");  }  }  class MyDerivedClass : MyBaseClass  {  override public void Print()  {  Console.WriteLine("This is the derived class.");  }  }  class Program  {  static void Main()  {  MyDerivedClass derived = new MyDerivedClass();  MyBaseClass mybc = (MyBaseClass)derived;  ↑  derived.Print(); مرجع گیری برای کلاس پایه  mybc.Print();  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| This is the derived class.  This is the derived class. |

مواردی که مهم است درباره اصلاحگر های virtual و override بدانید به شرح زیر است:

* متد های بازنویس[[6]](#footnote-6) و متد های بازنویسی شده[[7]](#footnote-7) باید سطح دسترسی یکسانی داشته باشند. به عبارت دیگر، نمی تواند متد بازنویسی شده private و متد بازنویس public باشد.
* متدی را که به صورت استاتیک یا virtual تعریف نشده است را نمی توان بازنویسی[[8]](#footnote-8) کرد.
* متدها، خصوصیات، ایندکسر ها (که در فصل قبل بررسی شده اند) و رویدادها (که در فصل بعد بررسی خواهد شد) می توانند به صورت virtual و override تعریف شوند.

بازنویسی کردن متد بازنویسی شده

بازنویسی متدها می تواند در هر سطحی از ارث بری انجام شود.

* هنگامی که مرجع ای برای کلاس پایه استفاده می شود، در فراخوانی متد بازنویس، فراخوانی متد به نسخه مشتق شده در سلسله مراتب ارث بری منتقل می شود.
* اگر تعریف دیگری از این متد در سطح ارث بری وجود داشته باشد که با کلمه override برچسب گذاری نشده باشند، این متدها با این کار (فراخوانی متد کلاس پایه) فراخوانی نمی شوند.

به عنوان مثال، کد زیر سه کلاس MyBaseClass ، MyDerivedClass و SecondDerived را نشان می دهد که به صورت سلسله مراتب از یکدیگر ارث بری می کنند. هر سه کلاس متدی به نام Print و با امضای یکسان دارند. در کلاس MyBaseClass متد Print با اصلاحگر virtual تعریف شده است. در کلاس MyDerivedClass متد Print با اصلاحگر override تعریف شده است. متد Print در کلاس SecondDerived، می تواند با کلمه کلیدی override یا new تعریف شود. در ادامه هر دو مورد را بررسی خواهیم کرد.

|  |
| --- |
| class MyBaseClass // کلاس پایه  {  virtual public void Print()  { Console.WriteLine("This is the base class."); }  }  class MyDerivedClass : MyBaseClass // کلاس مشتق شده  {  override public void Print()  { Console.WriteLine("This is the derived class."); }  }  class SecondDerived : MyDerivedClass // کلاس مشتق شده  {  ... // در صفحات بعد تعیین شده است  } |

مورد 1: تعریف متد Print با کلمه کلیدی override

اگر متد Print در کلاس SecondDerived به صورت override تعریف شود. پس این متد همانطور که در شکل 9-7 نشان داده شده است، بازنویسی برای دو متد دیگر خواهد بود. اگر مرجع ای برای کلاس پایه استفاده شود و متد Print را فراخوانی کند، این فراخوانی متد به صورت زنجیره ای به متد کلاس SecondDerived منتقل می شود.

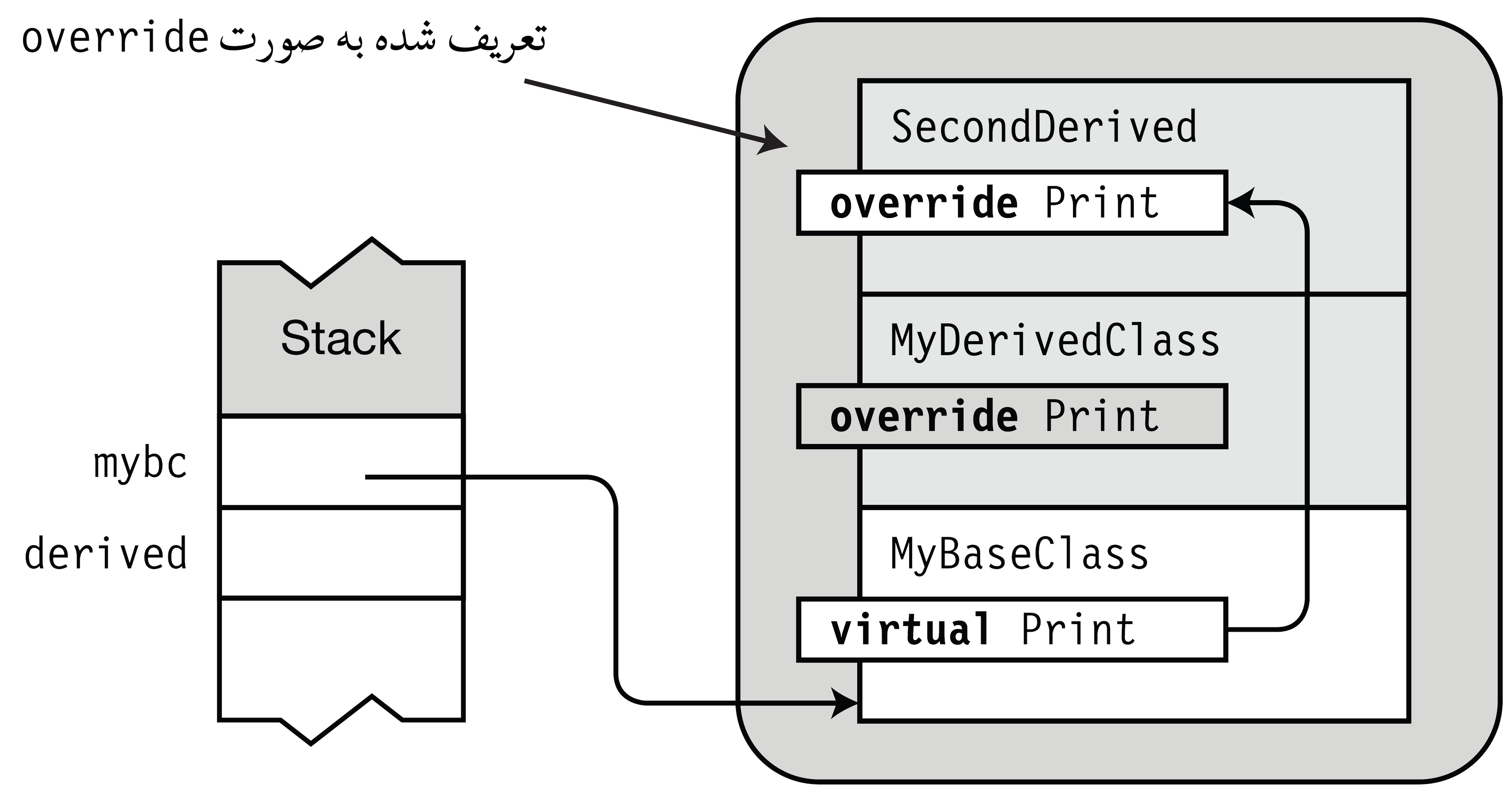
کد زیر این مورد را پیاده سازی کرده است. به دو خط آخر متد Main توجه کنید.

* اولین فراخوانی متد Print، توسط مرجع کلاس مشتق شده (SecondDerived) انجام می شود. این متد توسط کلاس پایه فراخوانی نمی شود، بنابراین متد کلاس SecondDerived فراخوانی می شود.
* دومین فراخوانی متد Print، توسط مرجع کلاس پایه (MyBaseClass) انجام می شود.

|  |
| --- |
| class SecondDerived : MyDerivedClass  {  override public void Print() {  ↑ Console.WriteLine("This is the second derived class.");  }  }  class Program  {  static void Main()  {  SecondDerived derived = new SecondDerived(); // SecondDerivedاستفاده از  MyBaseClass mybc = (MyBaseClass)derived; // MyBaseClassاستفاده از  derived.Print();  mybc.Print();  }  } |

نتیجه این است که صرف نظر از اینکه متد Print توسط مرجع کلاس مشتق شده فراخوانی شده است یا توسط کلاس پایه، در هر دو حالت متد کلاس مشتق شده فراخوانی شده است. هنگامی که متد Print از طریق مرجع کلاس پایه فراخوانی می شود این فراخوانی به سلسله مراتب ارث بری منتقل می شود. این کد خروجی زیر را تولید می کند:

|  |
| --- |
| This is the second derived class.  This is the second derived class. |



شکل 9-7. فراخوانی به بالاترین سطح زنجیره بازنویسی منتقل می شود.

مورد 2: تعریف متد Print با کلمه کلیدی new

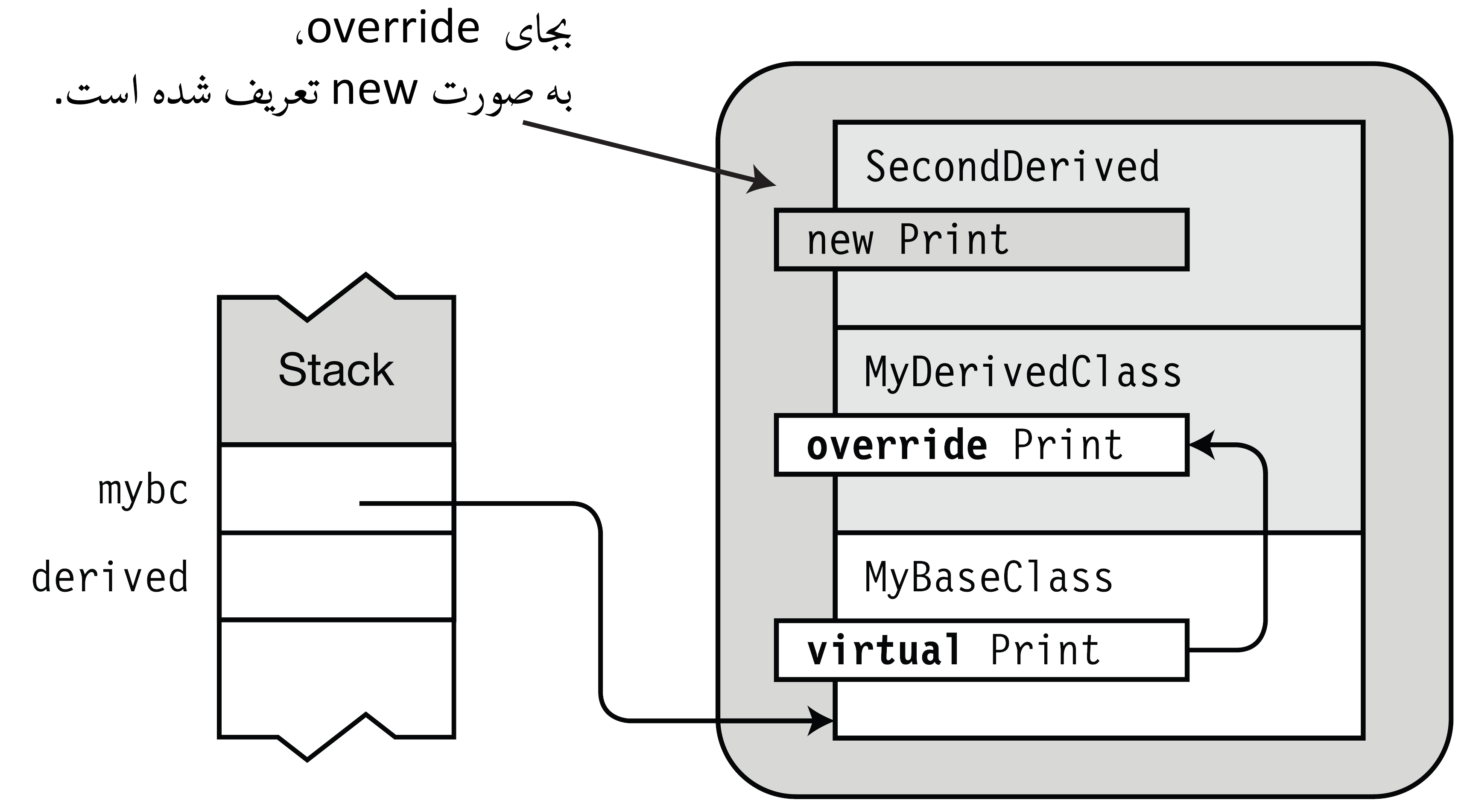
این بار متد Print در کلاس SecondDerived با کلمه کلیدی new تعریف شده است، نتیجه در شکل 10-7 نشان داده شده است. متد Main همان کد های استفاده شده در مورد اول را بکار می برد.

|  |
| --- |
| class SecondDerived : MyDerivedClass  {  new public void Print()  {  Console.WriteLine("This is the second derived class.");  }  }  class Program  {  static void Main() // Main متد  {  SecondDerived derived = new SecondDerived(); // SecondDerivedاستفاده از  MyBaseClass mybc = (MyBaseClass)derived; // MyBaseClassاستفاده از  derived.Print();  mybc.Print();  }  } |

نتیجه این است که زمانی که متد Print توسط مرجع کلاس SecondDerived فراخوانی شده است، همان متد کلاس SecondDerived اجرا شده است. هنگامی که این متد توسط مرجع کلاس MyBaseClass فراخوانی شده است، فراخوانی این متد به کلاس MyDerived منتقل شده است. تنها تفاوت بین این دو مورد در این است که آیا این متد در کلاس SecondDerived با کلمه کلیدی override تعریف شده است یا با کلمه کلیدی new.

این کد خروجی زیرا تولید می کند:

|  |
| --- |
| This is the second derived class.  This is the derived class. |



شکل 10-7. پنهان کردن متدهای بازنویسی شده

بازنویسی عضو های دیگر

در بخش قبل نحوه تعریف متدهای virtual و override را مشاهده کردید. این کار دقیقا برای خصوصیات و رویدادها و ایندکسر ها همانند متد ها است. به عنوان مثال، کد زیر خصوصیت فقط خواندنی MyProperty را نشان می دهد که از اصلاحگر های virtual و override استفاده می کند.

|  |
| --- |
| class MyBaseClass  {  private int \_myInt = 5;  virtual public int MyProperty  {  get { return \_myInt; }  }  }  class MyDerivedClass : MyBaseClass  {  private int \_myInt = 10;  override public int MyProperty  {  get { return \_myInt; }  }  }  class Program  {  static void Main()  {  MyDerivedClass derived = new MyDerivedClass();  MyBaseClass mybc = (MyBaseClass)derived;  Console.WriteLine( derived.MyProperty );  Console.WriteLine( mybc.MyProperty );  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| 10  10 |

اجرای سازنده

در فصل قبل نحوه اجرای سازنده ها را مشاهده کردید، که کلاس را برای استفاده آماده می کرد. این موضوع (آماده کردن کلاس) شامل مقداردهی اولیه عضو های استاتیک و عضو های نمونه است. در این فصل، خواهید دید که بخشی از شی کلاس مشتق شده، شی ای از کلاس پایه است.

* برای ایجاد بخش کلاس پایه شی، به طور ضمنی سازنده ای به عنوان بخشی از فرایند اجرا برای کلاس پایه فراخوانی می شود.
* هر کلاس در سلسله مراتب ارث بری، قبل از اجرا شدن بدنه سازنده خود، سازنده کلاس پایه خود را فراخوانی می کند.

به عنوان مثال، کد زیر کلاس MyDerivedClass را تعریف می کند. هنگامی که سازنده این کلاس فراخوانی می شود قبل از اجرا شدن سازنده، سازنده بدون پارامتر کلاس MyBaseClass فراخوانی می شود.

|  |
| --- |
| class MyDerivedClass : MyBaseClass  {  MyDerivedClass() // استفاده می کند MyBaseClass()این سازنده، از سازنده  {  ...  } |

شکل 11-7 ترتیب دستورات اجرایی در سازنده را نشان می دهد. اولین مواردی که در زمان اجرا انجام می شود، مقداردهی اولیه عضو های نمونه است. سپس سازنده کلاس پایه فراخوانی می شود. سپس سازنده کلاس اجرا می شود.



شکل 11-7. ترتیب دستورات اجرایی در سازنده

به عنوان مثال، در کد زیر، مقادیر فیلد هایMyField1 و MyField2 قبل از اجرای سازنده به ترتیب برابر 5 و 0 هستند.

|  |
| --- |
| class MyDerivedClass : MyBaseClass  {  int MyField1 = 5; // 1. مقدار دهی اولیه عضو  int MyField2; // مقدار دهی اولیه عضو  public MyDerivedClass() // 3. اجرای سازنده  {  ...  }  }  class MyBaseClass  {  public MyBaseClass() // 2. فراخوانی سازنده کلاس پایه  {  ...  }  } |

|  |
| --- |
| * احتیاط فراخوانی یک متد بازنویسی شده در سازنده توصیه نشده است. فراخوانی متد بازنویسی شده در کلاس پایه باعث انتقال فراخوانی به متد بازنویس در کلاس مشتق شده می شود در حالی که سازنده کلاس پایه در حال اجرا است. بنابراین متد کلاس مشتق شده قبل از مقداردهی اولیه اجرا می شود. |

مقدار دهنده سازنده

در زمانی که شی ای ساخته می شود، به طور پیش فرض، سازنده بدون پارامتر کلاس پایه آن فراخوانی می شود. به دلیل اینکه سازنده ها می تواند سربار گذاری شوند، بنابراین یک کلاس پایه ممکن است بیش از یک سازنده داشته باشد. اگر بخواهید کلاس مشتق شده ای را استفاده کنید، که از سازنده پارامتر دار کلاس پایه استفاده کند باید از مقدار دهنده سازنده[[9]](#footnote-9) استفاده کنید.

دو نوع مقدار دهنده سازنده وجود دارد:

* نوع اول از کلمه کلیدی base استفاده می کند و مشخص می کند کدام سازنده کلاس پایه باید استفاده شود.
* نوع دوم از کلمه کلیدی this استفاده می کند و مشخص می کند که باید از سازنده دیگری از این کلاس استفاده شود.

مقدار دهنده سازنده کلاس پایه به همراه دو نقطه بعد از تعریف سازنده کلاس قرار می گیرد. مقدار دهنده سازنده شامل کلمه کلیدی base و لیست پارامتر ها است.

به عنوان مثال، کد زیر سازنده ای از کلاس MyDerivedClass را نشان می دهد.

* مقدار دهنده سازنده مشخص می کند که در زمان ایجاد شی باید سازنده ای از کلاس پایه فراخوانی شود که به ترتیب دارای دو پارامتر از نوع string و int است.
* پارامترهای base باید از نظر نوع و تعداد با لیست پارامترهای سازنده کلاس پایه هماهنگ باشند.

|  |
| --- |
| مقدار دهنده سازنده  ↓  public MyDerivedClass( int x, string s ) : base( s, x )  { ↑  ... کلمه کلیدی |

هنگامی که سازنده ای بدون مقدار دهنده سازنده کلاس پایه تعریف می شود، این کار خلاصه شده ای از تعریف سازنده با مقدار دهنده سازنده base() است، هر دو نوع تعریف شده در شکل 12-7 معادل یک دیگر هستند.



شکل 12-7. دو سازنده معادل

شکل دیگر از دستور مقدار دهنده سازنده، فراخوانی سازنده های مختلفی از همان کلاس است. به عنوان مثال، در زیر سازنده ای تک پارامتر برای کلاس MyClass تعریف شده است. اگر سازنده تک پارامتر کلاس فراخوانی شود آنگاه سازنده دو پارامتر همان کلاس اجرا می شود که پارامتر دوم آن به صورت پیش فرض ارسال می شود.

|  |
| --- |
| مقدار دهنده سازنده  ↓  public MyClass(int x): this(x, "Using Default String")  { ↑  ... کلمه کلیدی  } |

وضعیت دیگری که در آن این کار مفید واقع می شود؛ زمانی است که یک کلاس باید چندین سازنده مختلف با کد های مشترک داشته باشد. که این کدها همیشه باید در زمان ایجاد شی اجرا شوند. در این حالت، می توان از کد های مشترک فاکتور گرفت و سازنده آن را به عنوان مقدار دهنده سازنده در سازنده های دیگر استفاده کرد. این عمل برای کاهش در تکرار کدها پیشنهاد شده است.

ممکن است فکر کنید که باید متد دیگری برای مقداردهی مشترک تعریف شود که تمام سازنده ها این متد را فراخوانی کنند. این به دلایلی خوب نیست. اول اینکه کامپایلر می تواند بعضی از موارد را بهینه سازی کند. دوم اینکه برخی از موارد تنها می تواند در یک سازنده انجام شود. به عنوان مثال، در فصل قبل آموختید که فیلد های readonly می توانند تنها در داخل سازنده مقداردهی اولیه شوند. اگر در متد دیگری برای مقداردهی یک فیلد readonly تلاش کنید کامپایلر پیغام خطا تولید می کند، حتی اگر این متد توسط سازنده فراخوانی شود.

اگر سازنده ای مشترک بتواند در خود کلاس به عنوان سازنده ای معتبر استفاده شود که تمام موارد مورد نیاز در کلاس را مقداردهی کند پس بهتر است که آن را از سازنده عمومی خارج کنیم.

با این حال، چی می شود اگر شی به طور کامل مقداردهی نشود؟ در این صورت، نباید اجازه دهید که سازنده از خارج از کلاس قابل فراخوانی باشد. زیرا شی به طور ناقص مقداردهی می شود. برای جلوگیری از این مشکل، می توان سازنده را بجای public به صورت private تعریف کرد و در تعریف سازنده های دیگر این سازنده را به عنوان مقدار دهنده سازنده استفاده کرد. کد زیر نحوه استفاده را نشان می دهد:

|  |
| --- |
| class MyClass  {  readonly int firstVar;  readonly double secondVar;  public string UserName;  public int UserIdNumber;  private MyClass( ) // انجام مقداردهی اولیه توسط سازنده خصوصی  { // برای سازنده های دیگر مشترک است  firstVar = 20;  secondVar = 30.5;  }  public MyClass( string firstName ) : this() // استفاده از مقدار دهنده سازنده  {  UserName = firstName;  UserIdNumber = -1;  }  public MyClass( int idNumber ) : this( ) // استفاده از مقدار دهنده سازنده  {  UserName = "Anonymous";  UserIdNumber = idNumber;  }  } |

اصلاحگر های دسترسی کلاس

یک کلاس می تواند توسط کلاس های دیگری قابل دسترسی باشد. در این بخش قابلیت دسترسی به کلاس را توضیح خواهیم داد. اگر چه می خواهیم از کلاس ها برای توضیحات این بخش استفاده کنیم اما این قوانین دسترسی برای نوع های دیگر نیز قابل اجرا است. که بعداً به بررسی آنها خواهیم پرداخت.

لفظ مشاهده پذیری و دیدن، گاهی اوقات برای لفظ دسترسی استفاده می شود. آنها را می توان مترادف یکدیگر دانست. دو سطح دسترسی public و internal برای کلاس وجود دارد.

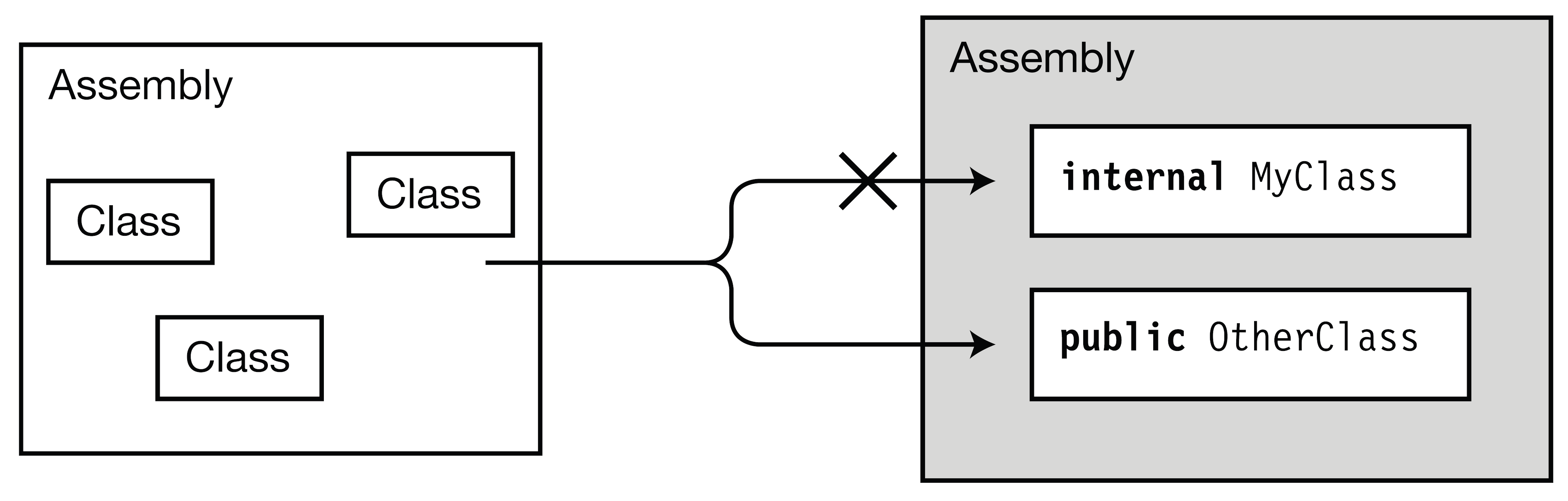
* یک کلاس public می تواند توسط های هر اسمبلی قابل دسترس باشد. همان طور که در زیر نشان داده شده است، برای ایجاد کلاسی که از سوی دیگر اسمبلی ها قابل مشاهده باشد، از اصلاحگر دسترسی public استفاده می شود.

|  |
| --- |
| کلمه کلیدی  ↓  public class MyBaseClass  { ... |

* یک کلاس internal می تواند تنها توسط کلاس های خود اسمبلی قابل دیدن باشد. (از فصل 1 به یاد دارید که یک اسمبلی یک برنامه یا یک DLL است. جزئیات را در فصل 21 بررسی خوایم کرد.)
  + این به طور پیش فرض تعریف می شود، بنابراین اگر به صراحت از اصلاحگر public در تعریف کلاس استفاده نشود، از خارج از اسمبلی نمی توان به کلاس دسترسی داشت.
  + می توان به صراحت از اصلاحگر internal در تعریف کلاس استفاده کرد.

|  |
| --- |
| کلمه کلیدی  ↓  internal class MyBaseClass  { ... |

شکل 13-7 دسترسی به کلاس های public و internal را از خارج از اسمبلی به تصویر کشیده است. کلاس MyClass برای کلاس های اسمبلی سمت چپ قابل مشاهده نیست، زیرا کلاس MyClass با اصلاحگر Internal تعریف شده است. کلاس OtherClass برای کلاس های اسمبلی سمت چپ قابل مشاهده است، زیرا با اصلاحگر public تعریف شده است.



شکل 13-7. کلاس های اسمبلی های دیگر می توانند به کلاس های public دسترسی داشته باشند، اما نمی توانند به کلاس های internal دسترسی داشته باشند.

ارث بری در بین اسمبلی ها

تا کنون، ما در تعریف کلاس های مشتق شده از کلاس پایه همان اسمبلی استفاده می کردیم. اما C# اجازه می دهد که از کلاس های اسمبلی های دیگر نیز ارث بری کنیم.

برای تعریف کلاس مشتق شده ای که از کلاس اسمبلی های دیگر ارث بری می کند باید موارد زیر رعایت شود:

* کلاس پایه باید به صورت public تعریف شده باشد که بتوان از خارج از اسمبلی به آن دسترسی داشت.
* باید مرجعی برای اسمبلی حاوی کلاس پایه به پروژه خود اضافه کنید.

برای اینکه آسان تر به کلاس ها و نوع های یک اسمبلی اشاره کنیم و همچنین نخواهیم از نام کامل آنها استفاده کنیم، کلمه کلیدی using را به همراه فضای نام آن اسمبلی استفاده می کنیم که در بالای کد منبع قرار می گیرد.

|  |
| --- |
| * توجه اضافه کردن یک مرجع به اسمبلی دیگر و اضافه کردن دستور using دو مبحث مجزا هستند. با اضافه کردن مرجع دیگر اسمبلی ها، به کامپایلر می گویند که نوع های مورد نیاز برنامه در این اسمبلی تعریف شده اند. اضافه کردن دستور using اجازه می دهد که مرجع کلاس های دیگر را بدون نام کامل آنها استفاده کنیم. این دو مبحث را در فصل 21 بررسی خواهیم کرد. |

به عنوان مثال، دو کد زیر که از اسمبلی های مختلفی هستند، نشان می دهد که چگونه از کلاس های موجود در اسمبلی های دیگر ارث بری می شود. کد های اولین اسمبلی شامل کلاس MyBaseClass است، که دارای ویژگی های زیر است:

* در فایل Assembly1.cs و در فضای نام BaseClassNS تعریف شده است.
* به صورت public تعریف شده است، بنابراین می توان از دیگر اسمبلی ها به آن دسترسی داشت.
* شامل متد PrintMe است که یک پیام ساده برای شناسایی کلاس چاپ می کند.

|  |
| --- |
| // Assembly1.cs  using System;  تعریف فضای نامی که حاوی کلاس پایه است  ↓  namespace BaseClassNS  { تعریف شده است بنابراین می توان ار خارج از اسمبلی به آن دسترسی داشت public کلاس به صورت  ↓  public class MyBaseClass {  public void PrintMe() {  Console.WriteLine("I am MyBaseClass");  }  }  } |

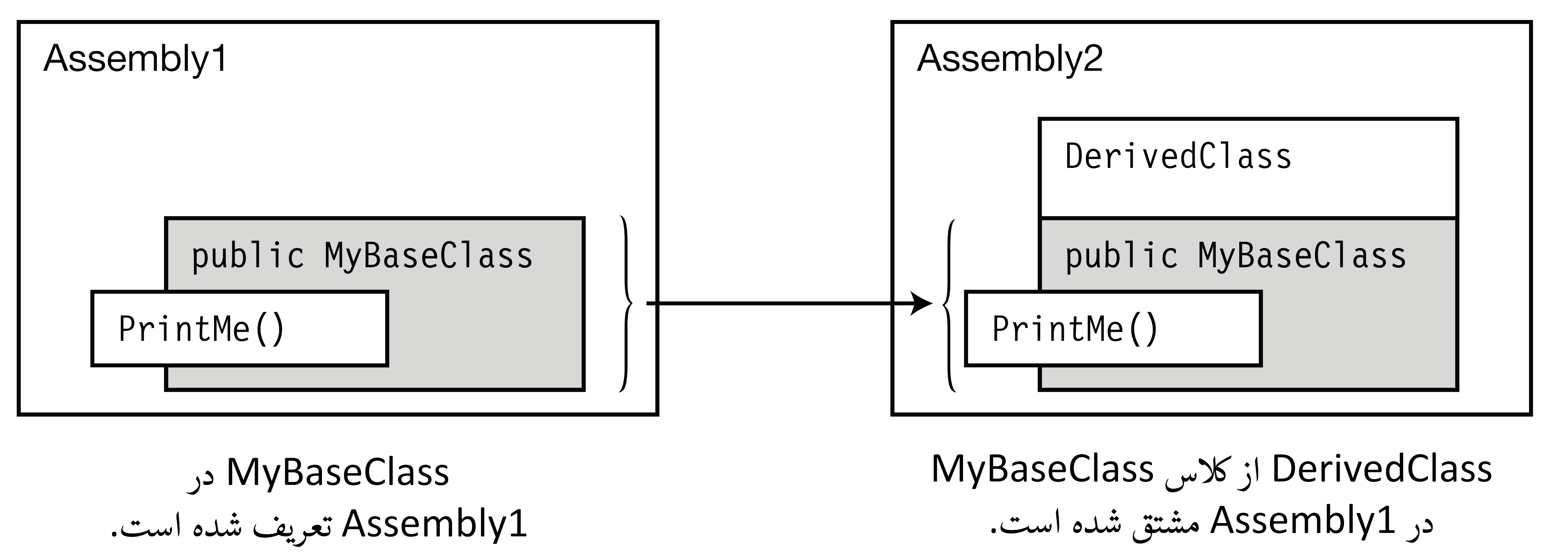
اسمبلی دوم شامل کلاس DerivedClass است که از کلاس MyBaseClass از اسمبلی اول ارث بری می کند. فایل منبع با نام Assembly2.cs نامگذاری شده است.شکل 14-7 هر دو اسمبلی را به تصویر کشیده است.

* کلاس DerivedClass بدنه خالی دارد اما متد PrintMe را از کلاس MyBaseClass به ارث می برد.
* متد Main شی ای از نوع DerivedClass ایجاد می کند و متد به ارث رسیده را فراخوانی می کند.

|  |
| --- |
| // Assembly2.cs  using System;  using BaseClassNS;  ↑  اعلان فضای نام شامل کلاس پایه  namespace UsesBaseClass  { کلاس پایه ای از اسمبلی دیگر  ↓  class DerivedClass: MyBaseClass {  // بدنه کلاس خالی است  }  class Program {  static void Main( )  {  DerivedClass mdc = new DerivedClass();  mdc.PrintMe();  }  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| I am MyBaseClass |



شکل 14-7. ارث بری در بین اسمبلی ها

اصلاحگر های دسترسی عضو

در دو بخش قبل اصلاحگر های دسترسی کلاس را توضیح دادیم که اصلاحگر های دسترسی کلاس شامل دو اصلاحگر public و internal است. در این بخش اصلاحگر های دسترسی عضو های کلاس را بررسی می کنیم. اصلاحگر های دسترسی کلاس قابلیت دیده شدن کلاس را توصیف می کند؛ و اصلاحگر های دسترسی عضو قابلیت دیده شدن عضو های کلاس را توصیف می کند.

هر عضو تعریف شده در کلاس برای قسمت های مختلف سیستم با توجه به اصلاحگری که برای آن عضو تعریف شده است قابل مشاهده است. دیدید که اعضای private فقط برای عضو های همان کلاس قابل مشاهده است؛ درحالی که عضو های public را می توان از خارج از کلاس مشاهده کرد. در این بخش، دوباره اصلاحگر دسترسی public و private و سه اصلاحگر دیگر را بررسی می کنیم.

قبل از پرداختن به جزئیات اصلاحگر های دسترسی، نیاز است که به موارد زیر توجه کنید:

* تمامی عضو ها تعریف شده کلاس صرف نظر از نوع اصلاحگر آنها، برای عضو های همان کلاس قابل مشاهده اند.
* عضو های به ارث رسیده از کلاس پایه به صراحت در کلاس مشتق شده تعریف نمی شوند. بنابراین خواهید دید که ممکن است عضو های به ارث رسیده از کلاس پایه برای عضو های کلاس مشتق شده قابل مشاهده باشند یا قابل مشاهده نباشند.
* در زیر اسامی اصلاحگر های دسترسی عضو را مشاهده می کنید. تا کنون، تنها اصلاحگر های public و private را معرفی کرده ایم.
  + public
  + private
  + protected
  + internal
  + protected internal
* باید سطح دسترسی هر عضو مشخص باشد. اگر سطح دسترسی یک عضو مشخص نشود، سطح دسترسی آن عضو به طور ضمنی private در نظر گرفته می شود.
* یک عضو نمی تواند بیشتر از کلاس خود قابل دسترسی باشد. به این معنا که اگر یک کلاس دارای سطح دسترسی internal باشد عضو های این کلاس نمی توانند از خارج از اسمبلی دیده شوند، حتی اگر سطح دسترسی عضوها به صورت public تعریف شده باشند.

مناطق دسترسی یک عضو

یک کلاس نسبت به اصلاحگر دسترسی عضو های خود مشخص می کند که کدام یک از عضو های آن می توانند توسط کلاس های دیگر قابل دسترسی باشند. تا کنون اصلاحگر های private و public را دیدید. دستور زیر کلاسی را تعریف می کند که پنج فیلد با اصلاحگر های دسترسی متفاوت دارد:

|  |
| --- |
| public class MyClass  {  public int Member1;  private int Member2;  protected int Member3;  internal int Member4;  protected internal int Member5;  ... |

کلاس دیگری (مثلا کلاس B) می تواند به این عضوها بر اساس دو مشخصه آن که به شرح زیر است، دسترسی داشته باشد.

* آیا کلاس B از کلاس MyClass مشتق شده است.
* آیا کلاس B در همان اسمبلی است که کلاس Myclass در آن است (آیا دو کلاس در یک اسمبلی هستند).

همانطور که در شکل 15-7 نشان داده شده است، از این دو مشخصه چهار گروه ایجاد می شود. با توجه به کلاس MyClass، کلاس های دیگر می توانند یکی از موارد زیر باشند:

* در همان اسمبلی و از کلاس MyClass مشتق شده باشد. ( پایین سمت راست)
* در همان اسمبلی باشد و از کلاس MyClass مشتق نشده باشد. ( پایین سمت چپ)
* در اسمبلی دیگری و از کلاس MyClass مشتق شده باشد. (بالا سمت راست)
* در اسمبلی دیگری باشد و از کلاس MyClass مشتق نشده باشد. (بالا سمت چپ)

این مشخصه ها برای تعریف پنج سطح دسترسی استفاده می شود، که در بخش بعدی آنها را بررسی خواهیم کرد.



شکل 15-7. مناطق دسترسی

دسترسی به عضو های public

سطح دسترسی public[[10]](#footnote-10) کمترین محدودیت است. تمام کلاس های داخل و خارج از اسمبلی می توانند به این عضو ها دسترسی پیدا کنند. شکل 16-5 قابلیت دسترسی به عضو های public از کلاس MyClass را به تصویر کشیده است.

همانطور که نشان داده شده است، برای تعریف یک عضو عمومی از اصلاحگر دسترسی public استفاده می شود.

|  |
| --- |
| کلمه کلیدی  ↓  public int Member1; |



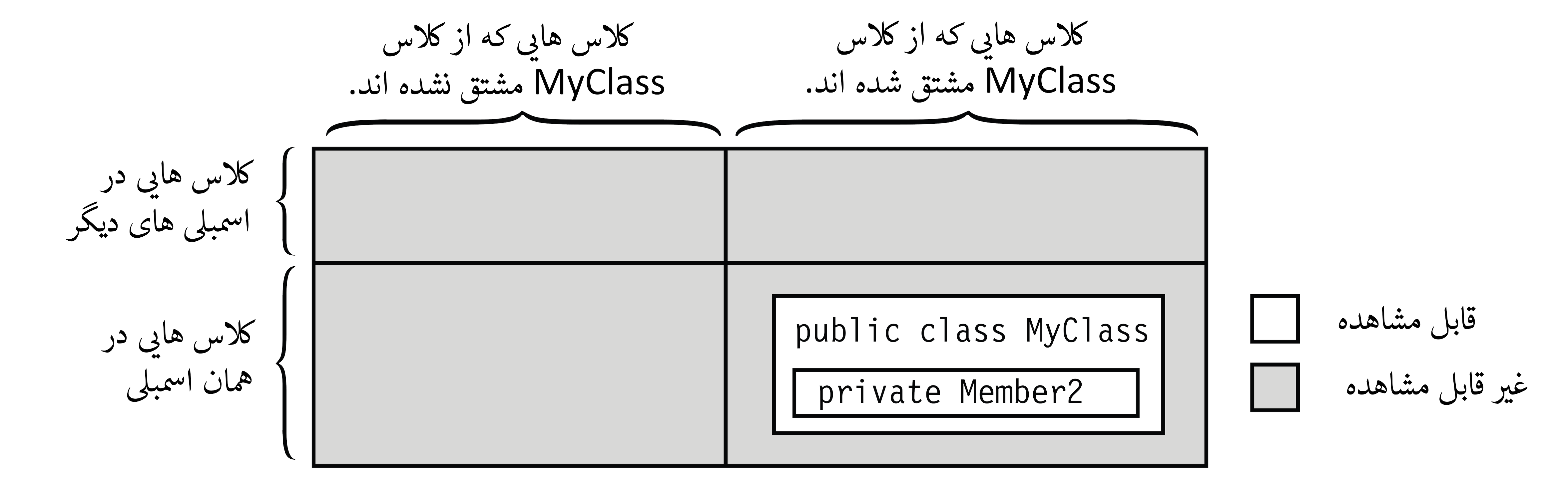
شکل 16-7. یک عضو عمومی از یک کلاس عمومی برای تمام کلاس های همان اسمبلی و دیگر اسمبلی ها قابل مشاهده است.

دسترسی به عضو های private

سطح دسترسی private[[11]](#footnote-11) بالاترین محدودیت است.

* یک عضو private فقط می تواند توسط عضو های کلاس خود قابل دسترس باشد. این عضوها نمی توانند توسط دیگر کلاس ها قابل دسترس باشد، از جمله کلاس هایی که از آن مشتق شده اند.
* با این حال، یک عضو private می تواند توسط عضو های کلاس های تودرتو در کلاس خود دیده شود. کلاس های تودرتو در فصل 25 بررسی شده اند.

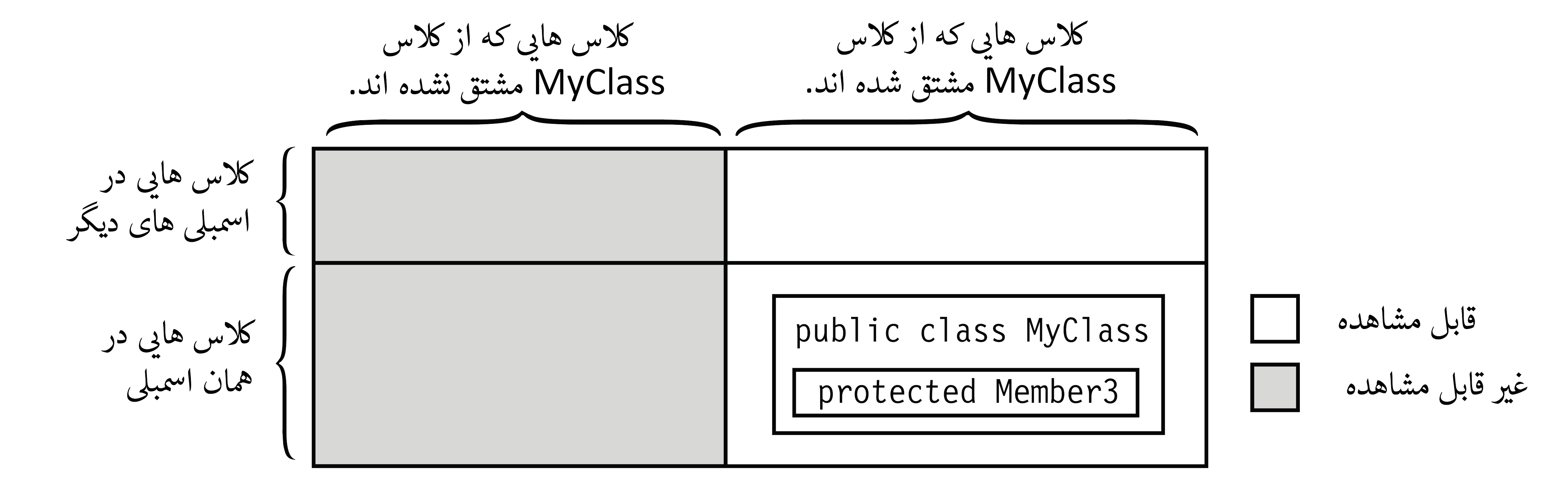
شکل 17-7 قابلیت دسترسی به یک عضو خصوصی را نشان می دهد.



شکل 17-7. یک عضو خصوصی از هر کلاس فقط برای عضو های کلاس خود (یا برای کلاس های تودرتو) قابل مشاهده است.

دسترسی به عضو های protected

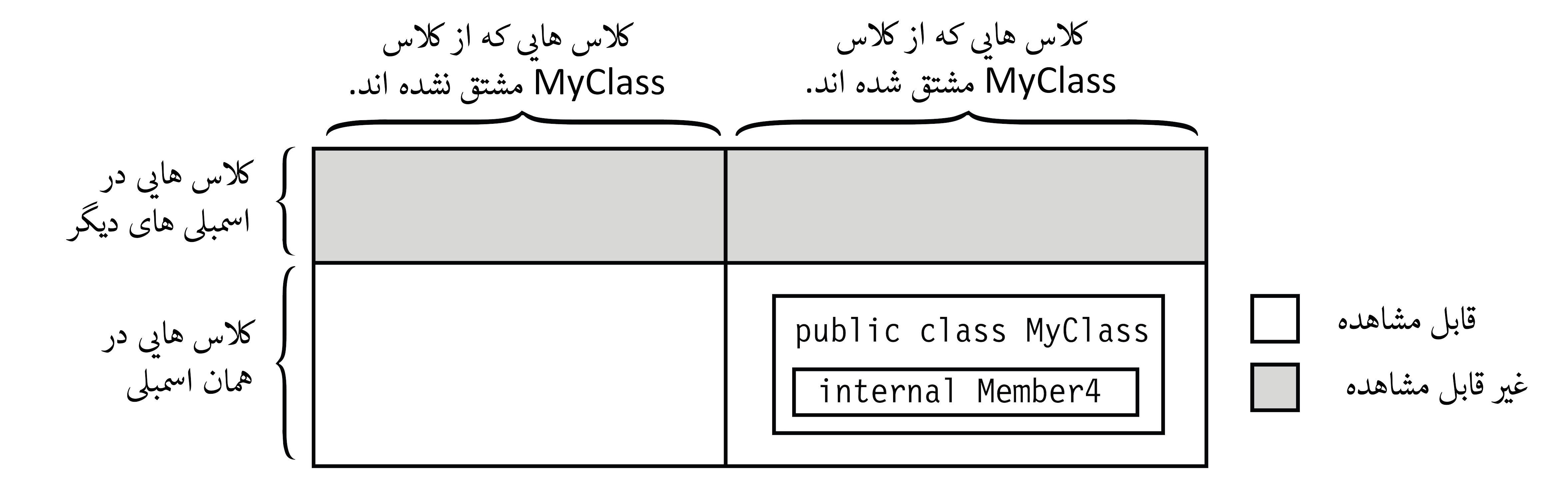
سطح دسترسی protected[[12]](#footnote-12) مانند سطح دسترسی private است، بجز این که اجازه می دهد که کلاس های مشتق شده به این عضو دسترسی داشته باشند. شکل 18-7 قابلیت دسترسی عضو های protected را به تصویر کشیده است. توجه کنید که حتی اگر کلاس در خارج از اسمبلی از این کلاس مشتق شود می تواند به عضو های protected دسترسی داشته باشد.



شکل 18-7. یک عضو محافظت شده از کلاس های عمومی برای عضو های کلاس خود و کلاس مشتق شده از آن قابل مشاهده است. حتی کلاس مشتق شده می تواند در اسمبلی دیگری باشد.

دسترسی به عضو های internal

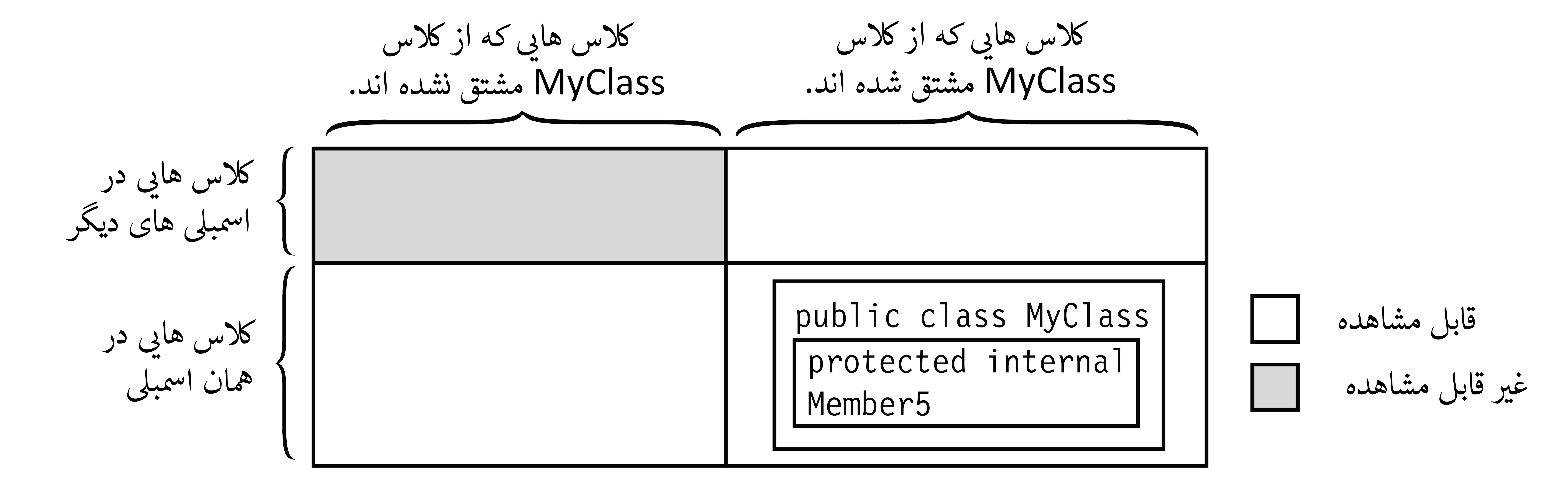
همانطور که در شکل 19-7 نشان میدهد، عضو های internal[[13]](#footnote-13) برای تمام کلاس های همان اسمبلی قابل مشاهده است، و برای کلاس های اسمبلی های دیگر قابل مشاهده نیست.



شکل 19-7. یک عضو داخلی از یک کلاس عمومی برای کلاس های همان اسمبلی قابل مشاهده است. اما برای کلاس های اسمبلی های دیگر قابل مشاهده نیست.

دسترسی به عضو های protected internal

همانطور که در شکل 20-7 نشان داده شده است، عضو های protected internal برای تمام کلاس هایی که از این کلاس ارث بری می کنند و برای تمام کلاس های موجود در همان اسمبلی قابل دسترس هستند. توجه کنید که مجموع کلاس هایی که اجازه دسترسی دارند ترکیبی از مجموعه کلاس های اصلاحگر protected و ترکیبی از مجموعه کلاس های اصلاحگر internal است. توجه کنید که مجموعه ای از protected و internal است و جدا از یکدیگر نیستند.



شکل 20-7. یک عضو محافظت شده داخلی در یک کلاس عمومی برای عضو های کلاس های همان اسمبلی و برای کلاس های مشتق شده از آن کلاس قابل مشاهده است. این برای کلاسی که در اسمبلی دیگری و از کلاس مشتق نشده است قابل مشاهده نیست.

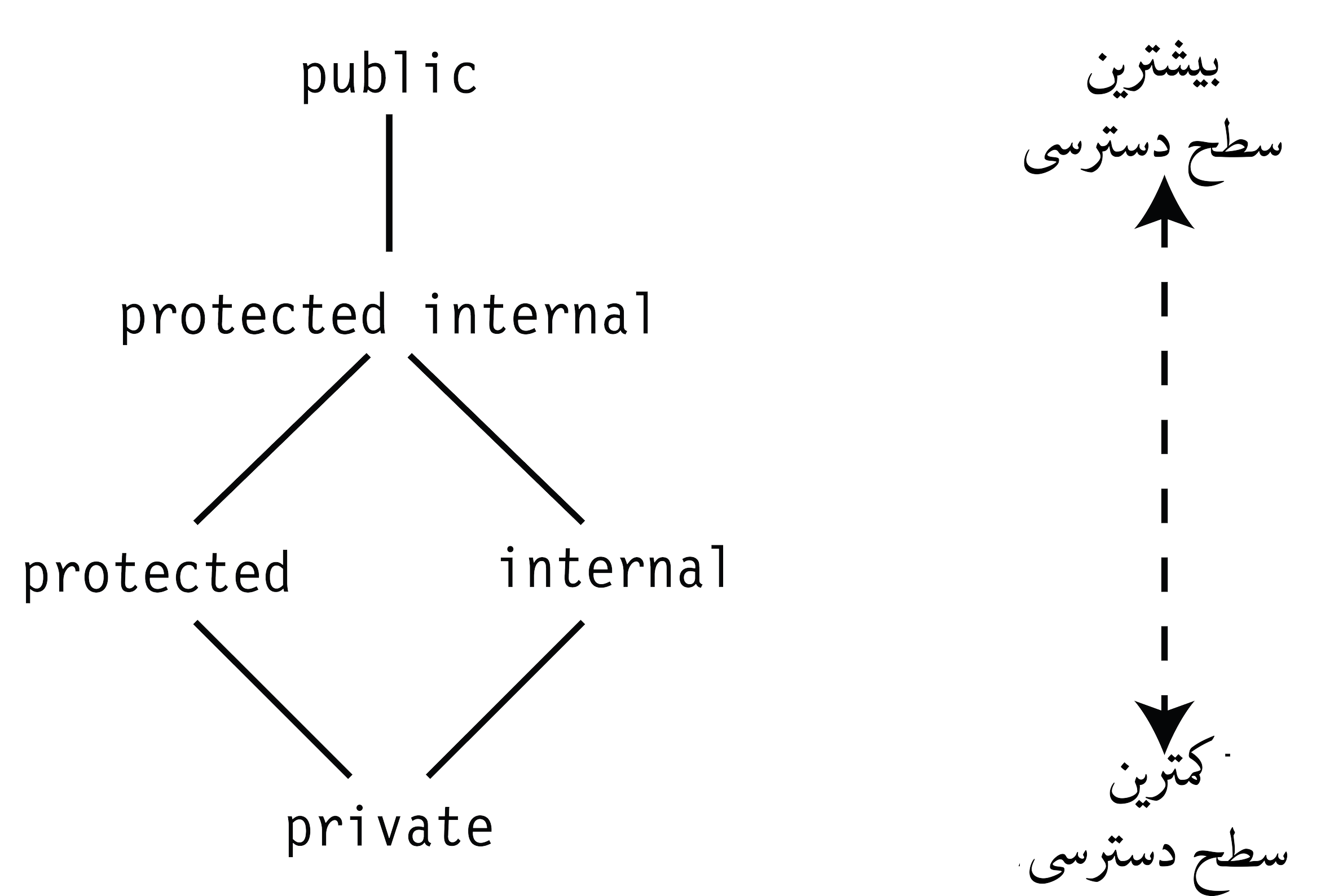
خلاصه ای از اصلاحگر های دسترسی عضو

دو جدول زیر مشخصه های پنج سطح دسترسی عضو را خلاصه کرده است. جدول 1-7 اصلاحگر های دسترسی و خلاصه ای از اثرات آنها را لیست کرده است.

جدول 1-7. اصلاحگر های دسترسی عضو

|  |  |
| --- | --- |
| اصلاحگر دسترسی | معنا |
| private | تنها برای همان کلاس قابل دسترس است |
| internal | برای تمام کلاس های همان اسمبلی قابل دسترس است |
| protected | برای تمام کلاس های مشتق شده از این کلاس قابل دسترس است |
| protected internal | برای تمام کلاس هایی که از این کلاس مشتق شده است و برای کلاس های همان اسمبلی قابل دسترس است |
| public | برای تمام کلاس های در دسترس است |

شکل 21-7 قابلیت دسترسی نسبی پنج اصلاحگر دسترسی عضو را نشان می دهد.



شکل 21-7. قابلیت دسترسی نسبی اصلاحگر های دسترسی عضو

جدول 2-7 اصلاحگر های دسترسی را در پایین و سمت راست جدول و گروهی از کلاس ها را در بالای جدول لیست کرده است. مشتق شده، به کلاس هایی اشاره می کند که از آن کلاس (کلاسی که عضو در آن تعریف شده است) مشتق می شوند. مشتق نشده، به کلاس هایی اشاره می کند که از آن کلاس مشتق نمی شوند. علامت تیک به این معنا است که این کلاس ها می توانند به اصلاحگر مربوطه دسترسی داشته باشند.

جدول 2-7. خلاصه ای از اصلاحگر های دسترسی عضو

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | کلاس هایی در همان اسمبلی | | | کلاس هایی در خارج از اسمبلی | | |
|  | مشتق نشده | مشتق شده |  | مشتق نشده | مشتق شده |  |
| private |  |  |  |  |  |  |
| internal |  |  |  |  |  |  |
| protected |  |  |  |  |  |  |
| protected internal |  |  |  |  |  |  |
| public |  |  |  |  |  |  |

عضو های انتزاعی

یک عضو انتزاعی[[14]](#footnote-14)، تابع عضوی است که برای بازنویسی کردن طراحی شده است. یک عضو انتزاعی ویژگی های زیر را دارد:

* باید یک تابع عضو باشد. به این معنا که، فیلد ها و ثابت ها نمی توانند عضو های انتزاعی باشند.
* باید با اصلاحگر abstract مشخص شده باشد.
* باید بلوکی از کد های اجرایی نداشته باشد. بلوک کد در عضو های انتزاعی با نقطه ویرگول مشخص می شود.

به عنوان مثال، کد زیر در داخل کلاس دو عضو انتزاعی PrintStuff و MyProperty را تعریف می کند. توجه کنید که از نقطه ویرگول بجای بلوک کد های اجرایی استفاده شده است.

|  |
| --- |
| کلمه کلیدی استفاده از نقطه ویرگول بجای بلوک کد های اجرایی  ↓ ↓  abstract public void PrintStuff(string s);  abstract public int MyProperty  {  get; ← استفاده از نقطه ویرگول بجای بلوک کد های اجرایی  set; ← استفاده از نقطه ویرگول بجای بلوک کد های اجرایی  } |

عضو های انتزاعی را تنها می توان در کلاس های انتزاعی تعریف کرد، که در بخش بعدی کلاس های انتزاعی را بررسی خواهیم کرد. چهار نوع عضو می توانند به صورت انتزاعی تعریف شوند:

* متد ها ـ Methods
* خصوصیات ـ Properties
* رویداد ها ـ Events
* ایندکسر ها ـ Indexers

مواردی که مهم است درباره عضو های انتزاعی بدانید به شرح زیر است:

* اگرچه عضو های انتزاعی را باید در کلاس مشتق شده بازنویسی کرد، اما اصلاحگر virtual را نمی توان در کنار اصلاحگر abstract استفاده کرد.
* مانند عضو های virtual، برای پیاده سازی یک عضو انتزاعی در کلاس مشتق شده باید آن را با اصلاحگر override مشخص کرد.

جدول 3-7 عضو های انتزاعی و عضو های virtual را مقایسه می کند.

جدول 3-7. مقایسه عضو های انتزاعی و عضو های virtual

|  |  |  |
| --- | --- | --- |
|  | عضو های virtual | عضو های انتزاعی |
| کلمه کلیدی | virtual | abstract |
| پیاده سازی بدنه | پیاده سازی بدنه دارد | پیاده سازی بدنه ندارد ــ بجای آن از نقطه ویرگول استفاده می شود |
| بازنویسی در یک کلاس مشتق شده | می تواند با استفاده از override بازنویسی شود | باید با استفاده از override بازنویسی شود |
| عضوها قابل تعریف | Methods  Properties  Events  Indexers | Methods  Properties  Events  Indexers |

کلاس های انتزاعی

کلاس های انتزاعی برای ارث بری طراحی شده اند. یک کلاس انتزاعی تنها می تواند به عنوان کلاس پایه ای برای کلاس های دیگر استفاده شود.

* نمی توان یک کلاس انتزاعی را نمونه سازی کرد.
* یک کلاس انتزاعی با اصلاحگر abstract تعریف می شود.

|  |
| --- |
| کلمه کلیدی  ↓  abstract class MyClass  {  ...  } |

* یک کلاس انتزاعی می تواند شامل عضو های انتزاعی باشد. عضو های یک کلاس انتزاعی می تواند هر ترکیبی از عضو های انتزاعی و عضو های عادی باشد.
* یک کلاس انتزاعی می تواند از کلاس انتزاعی دیگری مشتق شده باشد. به عنوان مثال، کد زیر نمونه ای از این ارث بری را نشان می دهد.

|  |
| --- |
| abstract class AbClass // کلاس انتزاعی  {  ...  }  abstract class MyAbClass : AbClass // کلاس مشتق شده انتزاعی از یک کلاس انتزاعی  {  ...  } |

* هر کلاس مشتق شده از کلاس انتزاعی باید تمام عضو های انتزاعی را با استفاده از کلمه کلیدی override پیاده سازی کند، مگر اینکه کلاس مشتق شده، انتزاعی باشد.

مثالی از کلاس های انتزاعی و عضو های انتزاعی

کد زیر کلاس انتزاعی AbClass را با دو متد نشان می دهد.

متد اول، متدی عادی است که کد اجرایی آن نام کلاس را چاپ می کند. متد دوم، متدی انتزاعی است که کد اجرایی آن باید در کلاس مشتق شده پیاده سازی شود. کلاس DerivedClass از کلاس AbClass ارث بری می کند و متد انتزاعی آن را با استفاده از کلمه کلیدی override پیاده سازی می کند. متد Main شی ای از کلاس DerivedClass ایجاد می کند و سپس هر دو متد آن را فراخوانی می کند.

|  |
| --- |
| کلمه کلیدی  ↓  abstract class AbClass // کلاس انتزاعی  {  public void IdentifyBase() // متد عادی  { Console.WriteLine("I am AbClass"); }  کلمه کلیدی  ↓  abstract public void IdentifyDerived(); // متد انتزاعی  }  class DerivedClass : AbClass // متد مشتق شده  { کلمه کلیدی  ↓  override public void IdentifyDerived() // پیاده سازی متد انتزاعی  { Console.WriteLine("I am DerivedClass"); }  }  class Program  {  static void Main()  {  // AbClass a = new AbClass (); // خطا. نمی توان یک کلاس انتزاعی  // a.IdentifyDerived(); // را نمونه سازی کرد.  DerivedClass b = new DerivedClass(); // نمونه سازی از کلاس مشتق شده  b.IdentifyBase(); // فراخوانی متد به ارث رسیده  b.IdentifyDerived(); // "abstract"فراخوانی متد  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| I am AbClass  I am DerivedClass |

مثال دیگری از کلاس های انتزاعی

کد زیر یک کلاس انتزاعی را نشان می دهد که شامل داده های عضو و توابع عضو است. به یاد داشته باشید که داده های عضو (فیلدها و ثابت ها) نمی توانند با اصلاحگر abstract تعریف شوند.

|  |
| --- |
| abstract class MyBase // ترکیبی از عضو های انتزاعی و غیر انتزاعی  {  public int SideLength = 10; // داده عضو  const int TriangleSideCount = 3; // داده عضو  abstract public void PrintStuff( string s ); // متد انتزاعی  abstract public int MyInt { get; set; } // خصوصیت انتزاعی  public int PerimeterLength( ) // متد غیر انتزاعی  { return TriangleSideCount \* SideLength; }  }  class MyClass : MyBase  {  public override void PrintStuff( string s ) // بازنویسی متد انتزاعی  { Console.WriteLine( s ); }  private int \_myInt;  public override int MyInt // بازنویسی خصوصیت انتزاعی  {  get { return \_myInt; }  set { \_myInt = value; }  }  }  class Program  {  static void Main( string[] args )  {  MyClass mc = new MyClass( );  mc.PrintStuff( "This is a string." );  mc.MyInt = 28;  Console.WriteLine( mc.MyInt );  Console.WriteLine( "Perimeter Length: {0}", mc.PerimeterLength( ) );  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| This is a string.  28  Perimeter Length: 30 |

کلاس های مهر و موم شده

در بخش قبل دیدید که یک کلاس انتزاعی باید به عنوان یک کلاس پایه استفاده شود و به طور مستقل نمی توان از آن نمونه سازی کرد. عکس این برای کلاس های مهر و موم شده[[15]](#footnote-15) صادق است.

* یک کلاس مهر و موم شده تنها می تواند به عنوان شی ای از یک کلاس مستقل نمونه سازی شده باشد و نمی تواند به عنوان کلاس پایه استفاده شود.
* یک کلاس مهر و موم شده با اصلاحگر sealed مشخص می شود.

به عنوان مثال، کلاس زیر یک کلاس مهر و موم شده است. اگر بخواهیم از این کلاس ارث بری کنیم، کامپایلر پیغام خطا تولید می کند.

|  |
| --- |
| کلمه کلیدی  ↓  sealed class MyClass  {  ...  } |

کلاس های استاتیک

کلاس استاتیک[[16]](#footnote-16)، کلاسی است که تمام عضو های آن استاتیک هستند. کلاس های استاتیک برای گروه و توابع هایی استفاده می شوند که تحت تاثیر نمونه سازی داده ها قرار نمی گیرد. ممکن است از کلاس های استاتیک برای ایجاد یک کتابخانه ریاضی استفاده شود، که شامل مجموعه ای از متدها و مقادیر ریاضی باشد.

مواردی که مهم است درباره کلاس های استاتیک بدانید به شرح زیر است:

* خود کلاس باید با کلمه کلیدی static مشخص شده باشد.
* تمام اعضای کلاس باید به صورت استاتیک تعریف شوند.
* این کلاس ها می توانند سازنده استاتیک داشته باشند، از آنجایی که نمی توان نمونه ای از این کلاس ها ایجاد کنید بنابراین نمی توان نمونه ای با این سازنده ایجاد کرد.
* کلاس های استاتیک به طور ضمنی مهر و موم شده هستند. به این معنا که، این کلاس ها نمی توانند به ارث برده شوند.

عضو های کلاس استاتیک فقط توسط نام کلاس و نام آن عضو قابل دسترس است.

کد زیر مثالی از کلاس های استاتیک را نشان می دهد:

|  |
| --- |
| مشخص شده باشد staticکلاس باید با کلمه کلیدی  ↓  static public class MyMath  {  public static float PI = 3.14f;  public static bool IsOdd(int x)  ↑ { return x % 2 == 1; }  عضوها باید به صورت استاتیک تعریف شده باشند  ↓  public static int Times2(int x)  { return 2 \* x; }  }  class Program  {  static void Main( )  { برای دسترسی به عضو از نام کلاس و نام عضو استفاده می شود  int val = 3; ↓  Console.WriteLine("{0} is odd is {1}.", val, MyMath.IsOdd(val));  Console.WriteLine("{0} \* 2 = {1}.", val, MyMath.Times2(val));  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| 3 is odd is True.  3 \* 2 = 6. |

متدهای الحاقی

هر متدی که تا کنون مشاهده کرده اید با کلاس تعریف شده خود در ارتباط بوده است. ویژگی های متد الحاقی اجازه می دهد، متد هایی نوشت که کلاس تعریف شده با کلاس های دیگر در ارتباط باشد.

به کد زیر نگاه کنید که چگونه می توان از این ویژگی ها استفاده کرد. این کد شامل کلاس MyData است که سه فیلد از نوع double را ذخیره می کند، و شامل یک سازنده و متد Sum است که مجموع سه رقم ذخیره شده را برمی گرداند.

|  |
| --- |
| class MyData  {  private double D1; // فیلد ها  private double D2;  private double D3;  public MyData(double d1, double d2, double d3) // سازنده  {  D1 = d1; D2 = d2; D3 = d3;  }  public double Sum() // Sumمتد  {  return D1 + D2 + D3;  }  } |

این کلاس بسیار محدود است، اما اگر شامل متد دیگری بود که میانگین این سه مقدار را برمی گرداند می توانست بسیار مفید تر باشد. با توجه به آنچه که تاکنون درباره کلاس ها آموختید، راه های مختلفی برای اضافه کردن این قابلیت وجود دارد:

* اگر کد منبع را داشته باشید می توان با اضافه کردن یک متد جدید کد این کلاس را تغییر داد.
* اگر به کد منبع دسترسی نداشته باشید (به عنوان مثال، اگر این کلاس در کتابخانه کلاسی شخص دیگری باشد)، پس تا زمانی که مهر و موم نشده باشد می توان آن را به عنوان کلاس پایه استفاده کرد و در کلاس مشتق شده این متد را اضافه کرد.

اگر به کد منبع دسترسی نداشته باشید و کلاس مهر و موم شده باشد یا دلایلی دیگر که مانع از انجام هر کدام از راه حل ها شود، بنابراین باید این متد را در کلاس دیگری پیاده سازی کنید که از عضو های عمومی این کلاس، استفاده کند.

به عنوان مثال، ممکن است کلاسی مانند کلاس زیر نوشته شود. که حاوی کلاس استاتیک ExtendMyData و این کلاس شامل متد Average باشد که قابلیت های اضافی را برای این کلاس پیاده سازی می کند. توجه کنید که پارامتر ورودی این متد نمونه ای از شی MyData است.

|  |
| --- |
| static class ExtendMyData MyData نمونه سازی از کلاس  { ↓  public static double Average( MyData md )  {  return md.Sum() / 3;  } ↑  } MyData استفاده از نمونه  class Program  {  static void Main()  { MyData نمونه ای از  MyData md = new MyData(3, 4, 5); ↓  Console.WriteLine("Average: {0}", ExtendMyData.Average(md));  } ↑  } فراخوانی متد استاتیک |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| Average: 4 |

اگر چه این راه حل بسیار مناسب است، اما اگر بجای نمونه سازی از یک کلاس دیگر بتوان این متد را با کلاس نمونه سازی شده خود فراخوانی کرد بسیار زیبا تر می شد. دو خط زیر تفاوت های او دو (فراخوانی با کلاس نمونه سازی خود و فراخوانی با کلاس دیگر) را نشان می دهد. اولین دستور، از متد استاتیک استفاده می کند که با کلاس نمونه سازی شده دیگری فراخوانی می شود. دومین دستور، دستور دلخواه ما را نشان می دهد، در این دستور، متد توسط کلاس نمونه سازی شده خود فراخوانی می شود.

|  |
| --- |
| ExtendMyData.Average( md ) // فراخوانی توسط کلاس استاتیک  md.Average(); // فراخوانی توسط کلاس نمونه سازی شده خود |

اگر چه شکل دستوری اول روشی عادی برای فراخوانی این متد است. اما متدهای الحاقی اجازه می دهند که از شکل دستوری دوم استفاده شود.

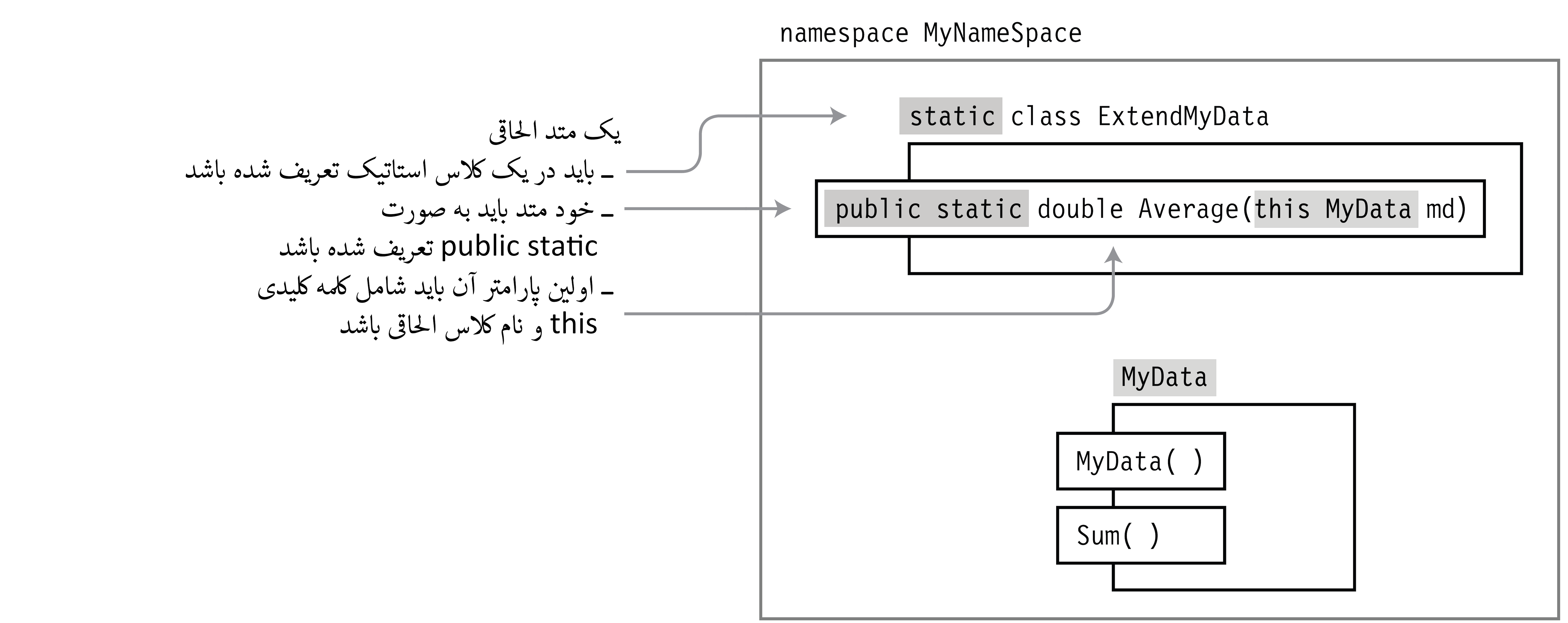
با ایجاد یک تغییر کوچک در تعریف متد، می توان از شکل دستوری دوم استفاده کرد. تغییراتی که باید انجام شود این است که قبل از نوع پارامتر ورودی از کلمه کلیدی this استفاده شود. زمانی که کلمه کلیدی this به پارامتر اول اضافه می شود، این متد می تواند توسط شی نمونه سازی شده کلاس MyData فراخوانی شود. برای فراخوانی این متد می توان از هر دو شکل دستوری استفاده کرد.

|  |
| --- |
| تعریف شود staticکلاس با به صورت  ↓  static class ExtendMyData  { باشد public و staticمتد باید به صورت کلمه کلیدی و نوع  ↓   ↓  public static double Average( this MyData md )  {  ...  }  } |

مواردی که مهم است درباره متدهای الحاقی بدانید به شرح زیر است:

* کلاسی که متد در آن تعریف شده است باید به صورت استاتیک تعریف شده باشد.
* خود متد الحاقی نیز باید به صورت استاتیک تعریف شود.
* اولین پارامتر از متد الحاقی باید شامل کلمه کلیدی this باشد، و سپس باید نام کلاس الحاقی آورده شود.

شکل 22-7 ساختار متد الحاقی را نشان می دهد.



شکل 22-7. ساختار متد الحاقی

کد زیر برنامه کاملی را نشان می دهد، که شامل کلاس MyData و متد بسط داده شده Average از کلاس ExtendMyData است. توجه کنید که به متد Average توسط نمونه ای از کلاس MyData فراخوانی شده است. شکل 22-7 این کد را به تصویر کشیده است. کلاس های MyData و ExtendMyData هر کدام به عنوان کلاسی دلخواه عمل می کنند.

|  |
| --- |
| namespace ExtensionMethods  {  sealed class MyData  {  private double D1, D2, D3;  public MyData(double d1, double d2, double d3)  { D1 = d1; D2 = d2; D3 = d3; }  public double Sum() { return D1 + D2 + D3; }  }  static class ExtendMyData کلمه کلیدی و نوع  { ↓  public static double Average(this MyData md)  { ↑  تعریف شده استstatic به صورت  return md.Sum() / 3;  }  }  class Program  {  static void Main()  {  MyData md = new MyData(3, 4, 5);  Console.WriteLine("Sum: {0}", md.Sum());  Console.WriteLine("Average: {0}", md.Average());  } ↑  } فراخوانی به عنوان عضوی از کلاس نمونه سازی شده  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| Sum: 12  Average: 4 |

قراردادهای نامگذاری

نوشتن برنامه به نام های بسیاری برای کلاس ها، متغییر ها، متد ها، خصوصیات، و چیزهایی که تاکنون بررسی نشده اند، نیاز دارد. هنگامی که در حال خواندن یک برنامه هستید، نامگذاری کنوانسیون شیوه ای مهم برای درک اشیایی است که با آنها مواجه می شوید.

در فصل 6 کمی نامگذاری را لمس کردید، اما در حال حاظر درباره کلاس های بیشتر می دانید، پس می توان جزئیات بیشتر را مطرح کرد. جدول 4-7 سه سبک اصلی نامگذاری و چگونگی آنها که معمولا در برنامه های دات نت مورد استفاده قرار می گیرد آورده شده است.

جدول 4-7. سبک های نامگذاری مورد استفاده

|  |  |  |  |
| --- | --- | --- | --- |
| نام سبک | تعریف | موارد استفاده | مثال |
| پوشش پاسگال[[17]](#footnote-17) | هر کلمه در شناسه با حروف بزرگ شروع می شود. | برای نام نوع ها و نام عضو ها که از خارج از کلاس قابل مشاهده است استفاده می شود. این موارد شامل موارد زیر است: کلاس ها، متد ها، namespace ها، خصوصیات، و فیلد های public. | CardDeck,  DealersHand |
| پوشش Camel | هر کلمه در شناسه، به جز اولین کلمه با حروف بزرگ شروع می شود. | برای نام متغییر های محلی و نام پارامتر های رسمی در تعریف متد استفاده می شود. | totalCycleCount,  randomSeedParam |
| حالت Camel با زیر خط[[18]](#footnote-18) | شناسه حالت Camel با یک زیر خط شروع شود. | برای نام فیلد های private و protected استفاده می شود. | \_cycleCount,  \_selectedIndex |

هر کسی با این کنوانسیون موافق نیست، به ویژه حالت زیر خط دار آن. خود من نیز با حالت زیر خط دار آن موافق نیستم، اما مفید است و آن را برای متغیرهای private و protected در کد خود استفاده کنید. خود مایکروسافت به نظر می رسد در این مسئله در تضاد باشد. در کنوانسیون های پیشنهادی آن، مایکروسافت شامل حالت زیر خط دار به عنوان یک گزینه نمی باشد. اما در بسیاری از کد ها خود از آن استفاده کرده است.

در بقیه این کتاب به کنوانسیون توصیه شده رسمی مایکروسافت برای استفاده از حالت زیر خط دار برای فیلد های private و protected پایبند خواهیم بود.

آخرین کلمه در مورد زیر خط این است که در بدنه یک شناسه استفاده نمی شود بجز در نام گرداننده رویداد ها[[19]](#footnote-19)، که در فصل 14 بررسی خواهند شد.

1. base class [↑](#footnote-ref-1)
2. derived class [↑](#footnote-ref-2)
3. single inheritance [↑](#footnote-ref-3)
4. base access [↑](#footnote-ref-4)
5. cast operator [↑](#footnote-ref-5)
6. متد هایی که از اصلاحگر override استفاده کرده اند. [↑](#footnote-ref-6)
7. متد هایی که از اصلاحگر virtual استفاده کرده اند. [↑](#footnote-ref-7)
8. override [↑](#footnote-ref-8)
9. constructor initializer [↑](#footnote-ref-9)
10. عمومی [↑](#footnote-ref-10)
11. خصوصی [↑](#footnote-ref-11)
12. محافظت شده [↑](#footnote-ref-12)
13. داخلی [↑](#footnote-ref-13)
14. abstract member [↑](#footnote-ref-14)
15. sealed classes [↑](#footnote-ref-15)
16. static class [↑](#footnote-ref-16)
17. Pascal casing [↑](#footnote-ref-17)
18. Camel case with leading underscore [↑](#footnote-ref-18)
19. Event handlers [↑](#footnote-ref-19)