فصل 8

عبارات و عملگر ها

* عبارات
* لیترال ها
* ترتیب ارزیابی
* عملگر های ریاضی
* عملگر باقی مانده
* عملگر های رابطه ای و تساوی
* عملگر های افزایش و کاهش
* عملگر های منطقی شرطی
* عملگر های منطقی
* عملگر های شیفت
* عملگر های انتساب
* عملگر انتساب شرطی
* عملگر های یگانی ریاضی
* تبدیل نوع های تعریف شده توسط کاربر
* سربار گذاری عملگر ها
* عملگر typeof
* دیگر عملگر ها

عبارات

این فصل، اصطلاحات و عملگر های ارائه شده در C# را بیان و توصیف می کند. همچنین توضیح می دهد که چگونه می توان عملگر هایی برای کلاس های تعریف شده توسط کاربر تعریف کرد.

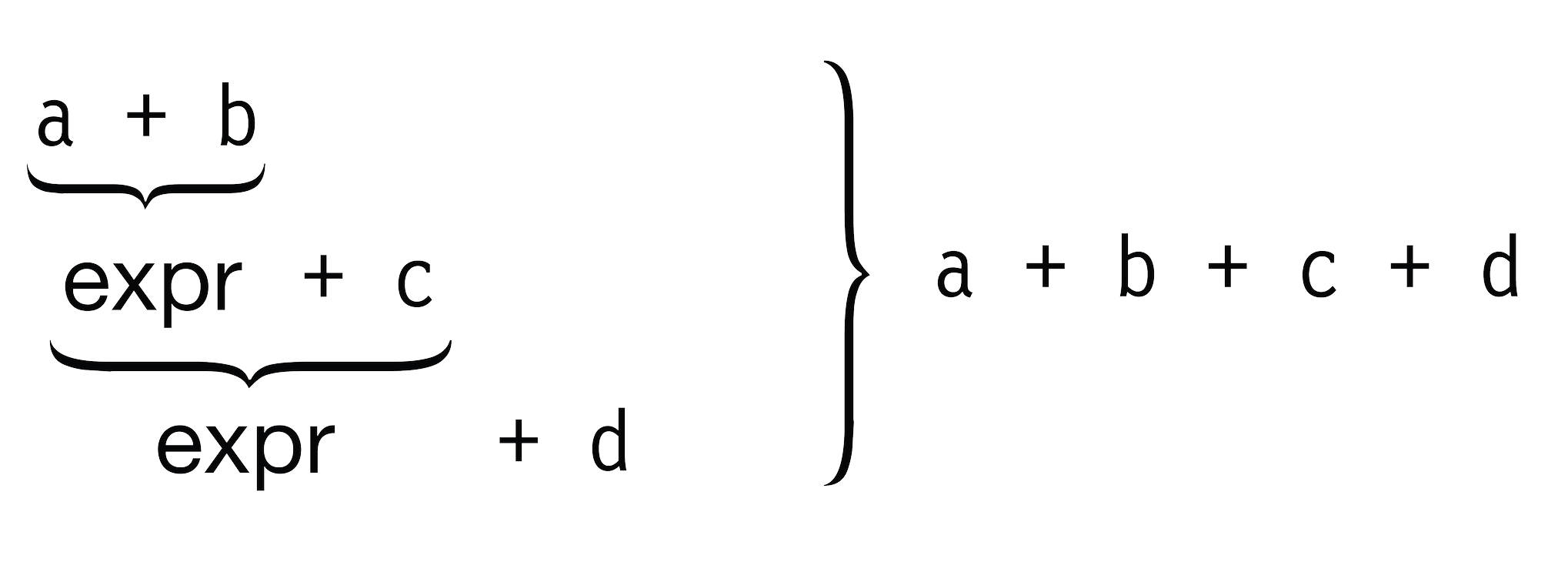
عملگر[[1]](#footnote-1) نمادی است که عملی را انجام می دهد و نتیجه آن را برمی گرداند. عملوند[[2]](#footnote-2) داده ای است که به عنوان ورودی توسط عملگر مورد استفاده قرار می گیرد. یک عملگر کارهای زیر را انجام می دهد:

* گرفتن ورودی های عملگر
* انجام عمل تعریف شده خود
* برگرداندن مقداری، بر اساس عمل انجام شده

عبارت[[3]](#footnote-3)، رشته ای از عملگر ها و عملوند ها است. عملگر های C# یک ، دو یا سه عملوند می گیرند. در زیر برخی از این ساختارها وجود دارد که می توانند به عنوان عملوند عمل کنند.

* لیترال ها
* ثابت ها
* متغییر ها
* فراخوانی متدها
* دسترسی به عناصر، مانند دسترسی به آرایه ها و ایندکسر ها
* دیگر عبارات

عبارات می توانند ترکیب شوند، و از عملگر ها می توان برای ایجاد عبارات پیچیده استفاده کرد، مانند عبارت زیر که از سه عملگر و چهار عملوند استفاده کرده است.



ارزیابی یک عبارت، فرآیندی در ترتیب و تولید مقدار از هر یک از عملوند های آن عملگر است.

* مقدار به موقعیت عبارت مورد بررسی بازگردانده می شود. این مقدار ممکن است به عنوان عملوندی برای عملگر دیگری استفاده شود.
* علاوه بر مقدار برگشت داده شده، برخی از عبارات اثرات جانبی[[4]](#footnote-4) دارند، مانند تنظیم یک مقدار در حافظه است.

لیترال ها

لیترال ها اعداد یا رشته های تایپ شده در کد منبع هستند که بیانگر مقدار خاصی از یک نوع خاص است.

به عنوان مثال، کد زیر شش نوع از لیترال ها را نشان می دهد. به عنوان مثال، به تفاوت بین لیترال double و لیترال float توجه کنید.

|  |
| --- |
| static void Main() لیترال ها  { ↓  Console.WriteLine("{0}", 1024); // int لیترال  Console.WriteLine("{0}", 3.1416); // double لیترال  Console.WriteLine("{0}", 3.1416F); // float لیترال  Console.WriteLine("{0}", true); // boolean لیترال  Console.WriteLine("{0}", 'x'); // character لیترال  Console.WriteLine("{0}", "Hi there"); // string لیترال  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| 1024  3.1416  3.1416  True  x  Hi there |

به دلیل اینکه لیترال ها در کد منبع نوشته می شوند، مقدار آنها در زمان کامپایل باید مشخص باشد. تعدادی از نوع های از پیش تعریف شده، لیترال خاص خود را دارند:

* نوع bool دارای دو لیترال true و false است.
* برای متغییر هایی از نوع ارجاعی، لیترال null بدان معنا است که متغییر به داده ای در حافظه اشاره نمی کند.

لیترال های اعداد صحیح

لیترال های اعداد صحیح رایج ترین لیترال هایی هستند که استفاده می شوند. آنها مانند دنباله ای از اعداد دهدهی با ویژگی های زیر نوشته می شوند:

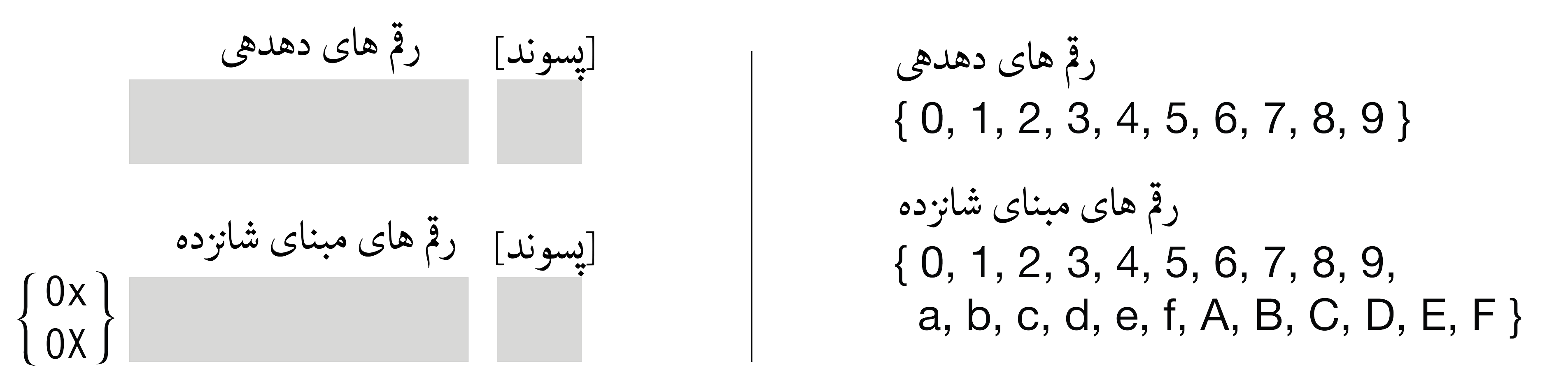
* بدون قسمت اعشاری
* پسوندی اختیاری برای مشخص کردن نوع عدد صحیح

به عنوان مثال، خطوط زیر چهار لیترال از عدد صحیح 236 را نشان می دهد. کامپایلر بسته به نوع پسوند آنها، نوع های متفاوتی را برداشت می کند.

|  |
| --- |
| 236 // int  236L // long  236U // unsigned int  236UL // unsigned long |

همچنین لیترال اعداد صحیح می توانند در مبنای شانزده (hex) نوشته شوند. رقم آن باید بین 0 تا F باشد، و این رشته باید با 0x یا 0X آغاز شده باشد.

شکل 1-8 شکل هایی از فرمت های لیترال را نشان می دهد. قسمت براکت اختیاری است.



شکل 1-8. فرمت های لیترال اعداد صحیح

جدول 1-8 پسوندهای لیترال اعداد صحیح را لیست کرده است. برای پسوند داده شده، کامپایلر کوچک ترین نوعی را که این عدد می تواند در آن ذخیره شود، برداشت می کند.

به عنوان مثال، لیترال 236 و 5000000000 بدون پسوند هستند. از آنجایی که 236 را می توان در 32 بیت نمایش داد، کامپایلر برداشت می کند که این عدد از نوع int است. با این حال، عدد دوم نمی تواند در 32 بیت ذخیره شود بنابراین کامپایلر برداشت می کند که این عدد از نوع long است.

جدول 1-8. پسوند های لیترال اعداد صحیح

|  |  |  |
| --- | --- | --- |
| پسوند | نوع | نکات |
| بدون پسوند | int, uint, long, ulong |  |
| U و u | uint, ulong |  |
| L و l | long, ulong | استفاده از حروف کوچک l توصیه نمی شود، زیرا به راحتی با رقم 1 اشتباه گرفته می شود. |
| UL و Ul و uL و ul  LU و lU و Lu و lu | ulong | استفاده از حروف کوچک l توصیه نمی شود، زیرا به راحتی با رقم 1 اشتباه گرفته می شود. |

لیترال اعداد واقعی

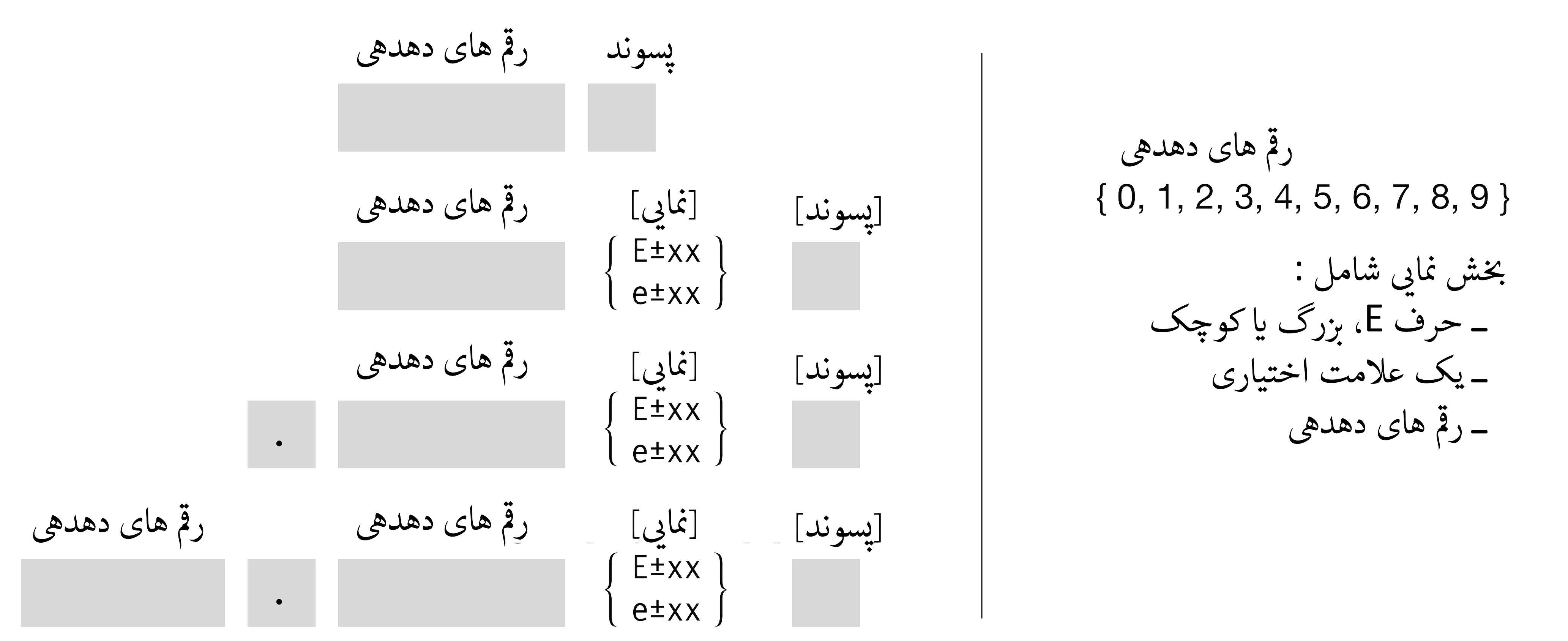
لیترال اعداد واقعی از موارد زیر تشکیل شده است:

* رقم های دهدهی
* علامت اعشار (اختیاری)
* بخش نمایی (اختیاری)
* پسوند (اختیاری)

به عنوان مثال، کد زیر فرمت های مختلفی از لیترال نوع های واقعی را نشان می دهد.

|  |
| --- |
| float f1 = 236F;  double d1 = 236.714;  double d2 = .35192;  double d3 = 6.338e-26; |

شکل 2-8 فرمت های معتبری برای لیترال اعداد واقعی را نشان می دهد. قسمت های براکت اختیاری هستند. جدول 2-8 پسوند اعداد واقعی را نشان می دهد.



شکل 2-8. فرمت های لیترال اعداد واقعی

جدول 2-8. پسوندهای لیترال اعداد واقعی

|  |  |
| --- | --- |
| پسوند | نوع |
| بدون پسوند | double |
| F و f | float |
| D و d | double |
| M و m | decimal |

|  |
| --- |
| * توجه لیترال اعداد واقعی بدون پسوند از نوع double هستند و از نوع float نیستند. |

لیترال کاراکتر ها

لیترال کاراکتر ها شامل یک کاراکتر محصور شده بین دو علامت نقل قول است. نماینده کاراکتر میتواند هر یک موارد زیر باشد: یک کاراکتر، یک ترتیب گریز[[5]](#footnote-5) ساده، ترتیب گریز مبنای شانزده، ترتیب گریز یونیکد.

* لیترال کاراکتر ها از نوع char هستند.
* یک ترتیب گریز ساده شامل بک اسلش و یک کاراکتر است.
* یک ترتیب گریز مبنای شانزده، شامل بک اسلش و کاراکتر X یا x و چهار رقم از اعداد مبنای شانزده است.
* یک ترتیب گریز یونیکد، شامل بک اسلش و کاراکتر U یا u و چهار رقم از اعداد مبنای شانزده است.

به عنوان مثال، کد زیر فرمت های مختلفی از لیترال کاراکتر ها را نشان می دهد.

|  |
| --- |
| char c1 = 'd'; // تک کاراکتر  char c2 = '\n'; // ترتیب گریز ساده  char c3 = '\x0061'; // ترتیب گریز مبنای شانزده  char c4 = '\u005a'; // ترتیب گریز یونیکد |

جدول 3-8 تعدادی از مهم ترین کاراکتر های خاص را نشان می دهد.

جدول 3-8. مهم ترین کاراکتر های خاص

|  |  |  |
| --- | --- | --- |
| نام | ترتیب گریز | کد مبنای شانزده |
| Null | \0 | 0x0000 |
| Alert | \a | 0x0007 |
| Backspace | \b | 0x0008 |
| Horizontal tab | \t | 0x0009 |
| New line | \n | 0x000A |
| Vertical tab | \v | 0x000B |
| Form feed | \f | 0x000C |
| Carriage return | \r | 0x000D |
| Double quote | \" | 0x0022 |
| Single quote | \' | 0x0027 |
| Backslash | \\ | 0x005C |

لیترال رشته ها

لیترال رشته ها از دو علامت نقل قول دوگانه استفاده می کند و از نقل قول یگانه استفاده نمی کند، نقل قول یگانه برای لیترال کاراکتر ها استفاده می شود. دو نوع لیترال برای رشته ها وجود دارد:

* لیترال رشته های منظم
* لیترال رشته های کلمه ای

یک لیترال منظم شامل دنباله ای از لیترال های کاراکتر در علامت نقل قول دوگانه است. یک لیترال رشته ای منظم می تواند شامل موارد زیر باشد:

* کاراکتر
* ترکیب گریز ساده
* ترکیب گریز مبنای شانزده و یونیکد

در زیر چند مثال وجود دارد:

|  |
| --- |
| string st1 = "Hi there!";  string st2 = "Val1\t5, Val2\t10";  string st3 = "Add\x000ASome\u0007Interest"; |

لیترال رشته های کلمه ای مانند لیترال رشته های منظم نوشته می شوند اما با کاراکتر @ آغاز می شوند. مهمترین ویژگی لیترال رشته های کلمه ای به شرح زیر است:

* لیترال رشته های کلمه ای در اینکه "ترتیب گریز ها" مورد ارزیابی قرار می گیرند یا نه، با لیترال رشته های منظم تفاوت دارند، در لیترال رشته های کلمه ای تمام موارد بین نقل قول ها همانطوری است که در رشته قرار گرفته است.
* تنها استثنا در لیترال کلمه ای، علامت نقل قول است که به عنوان کاراکتر نقل قول در باز و بسته کردن رشته تفسیر می شود.

به عنوان مثال، کد زیر برخی از لیترال های منظم و کلمه ای را مقایسه می کند:

|  |
| --- |
| string rst1 = "Hi there!";  string vst1 = @"Hi there!";  string rst2 = "It started, \"Four score and seven...\"";  string vst2 = @"It started, ""Four score and seven...""";  string rst3 = "Value 1 \t 5, Val2 \t 10"; // تفسیر می شوند (tab)به عنوان تب \t  string vst3 = @"Value 1 \t 5, Val2 \t 10"; // تفسیر نمی شوند (tab)به عنوان تب \t  string rst4 = "C:\\Program Files\\Microsoft\\";  string vst4 = @"C:\Program Files\Microsoft\";  string rst5 = " Print \x000A Multiple \u000A Lines";  string vst5 = @" Print  Multiple  Lines"; |

چاپ این رشته ها خروجی زیر را تولید می کند:

|  |
| --- |
| Hi there!  Hi there!  It started, "Four score and seven..."  It started, "Four score and seven..."  Value 1 5, Val2 10  Value 1 \t 5, Val2 \t 10  C:\Program Files\Microsoft\  C:\Program Files\Microsoft\  Print  Multiple  Lines  Print  Multiple  Lines |

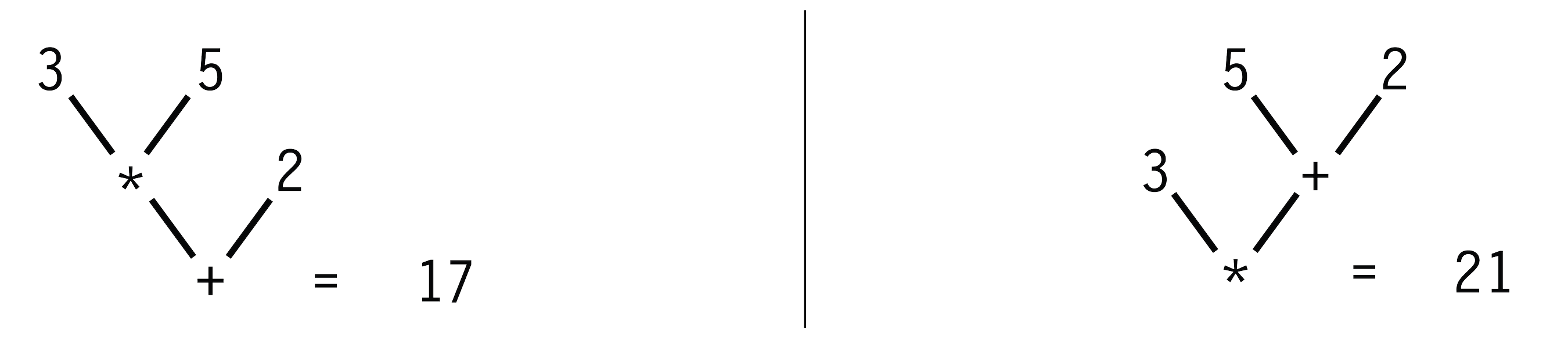
|  |
| --- |
| * توجه با داشتن لیترال رشته های یکسان، کامپایلر برای صرفه جویی در حافظه همان مکان از حافظه heap را به اشتراک می گذارد. |

ترتیب ارزیابی

یک عبارت می تواند از مجموعه ای از عبارات تو در تو ساخته شده باشد. ترتیب ارزیابی این مجموعه عبارات می تواند مقدار متفاوتی را تولید کند.

به عنوان مثال، همانطور که در شکل 3-8 نشان داده شده است، عبارت 3 \* 5 + 2، بسته به ترتیب آنها ممکن است دو نتیجه متفاوت تولید کند.

* اگر ابتدا ضرب انجام شود، نتیجه آن 17 خواهد شد.
* اگر ابتدا 2 و 5 با یکدیگر جمع شوند، نتیجه آن 21 خواهد شد.



شکل 3-8. مثالی از ترتیب ارزیابی

اولویت

از دوران مدرسه می دانید که در مثال قبل، ضرب باید قبل از جمع انجام شود، زیرا عملگر ضرب دارای اولویت بالاتری نسبت به عملگر جمع است. اما برخلاف دوران مدرسه که چهار عملگر و دو سطح اولویت داشتید، با C# همه چیز کمی پیچیده تر است، C# بیش از 45 نوع عملگر و 14 نوع سطح اولویت دارد.

جدول 4-8 لیست کاملی از عملگر ها و اولویت ها را نشان می دهد. بالای جدول، بیشتر اولویت و پایین جدول، کمترین اولویت را دارند.

جدول 4-8. اولویت عملگر ها: از بیشترین اولویت به کمترین اولویت

|  |  |
| --- | --- |
| دسته ها | عملگر ها |
| نخستین | a.x, f(x), a[x], x++, x--, new, typeof, checked, unchecked |
| یگانی | +, -, !, ~, ++x, --x, (T)x |
| ضربی | \*, /, % |
| افزودنی | +, - |
| انتقال | <<, >> |
| رابطه و نوع | <, >, <=, >=, is, as |
| تساوی | ==, != |
| AND منطقی | & |
| XOR منطقی | ^ |

|  |  |
| --- | --- |
| دسته ها | عملگر ها |
| OR منطقی | | |
| AND شرطی | && |
| OR شرطی | || |
| انتساب شرطی | ?: |
| تخصیص | =, \*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |= |

شرکت پذیری

زمانی که کامپایلر یک عبارت که در آن سطوح مختلفی از اولویت عملگر ها در آن قرار دارد را ارزیابی می کند. ابتدا از بالاترین سطح اولویت شروع به ارزیابی می کند.

ابتدا کدام یک از عملگر هایی که به طور متوالی و در یک سطح از اولویت قرار دارد ارزیابی می شود؟ به عنوان مثال، در عبارت 2 / 6 \* 4 دو نوع ارزیابی وجود دارد:

(2 / 6) \* 4 = 4/3

یا

2 / (6 \* 4) = 1/12

هنگامی که عملگر های متوالی در همان سطح از اولویت وجود دارند، ترتیب ارزیابی توسط شرکت پذیری عملگر تعیین می شود. به این معنا که دو عملگر داده شده در همان سطح از اولویت، بسته با شرکت پذیری عملگر ها، یکی بر دیگری اولویت دارد. برخی از ویژگی های مهم شرکت پذیری عملگر ها به شرح زیر است و در جدول 5-8 خلاصه شده است:

* شرکت پذیر از چپ، عملگر هایی که از سمت چپ به راست ارزیابی می شوند.
* شرکت پذیر از راست، عملگر هایی که از سمت راست به چپ ارزیابی می شوند.
* عملگر های باینری، بجز عملگر های انتساب، شرکت پذیر چپ هستند.
* عملگر های انتساب، عملگر های شرطی شرکت پذیر راست هستند.

بنابراین، با توجه به این قوانین، عبارت مثال قبل باید از سمت چپ به راست به (2 / 6) \* 4 گروه بندی شده باشد که جواب آن 4/3 خواهد شد.

جدول 5-8. خلاصه ای از شرکت پذیری عملگر ها

|  |  |
| --- | --- |
| نوع عملگر | شرکت پذیری |
| عملگر انتساب | شرکت پذیر از راست |
| سایر عملگر های باینری | شرکت پذیر از چپ |
| عملگر های شرطی | شرکت پذیر از راست |

با استفاده از پرانتز به صراحت می توان ترتیب ارزیابی عبارات را تعیین کرد.

* نادیده گرفتن اولویت و قوانین شرکت پذیری
* به ترتیب از پرانتز های تو در تو درونی به بیرونی ترین آنها ارزیابی می شود.

عملگرهای ریاضی

عملگر های محاسباتی ساده، چهار عملیات ریاضی پایه را انجام می دهد و در جدول 6-8 ذکر شده اند. این عملگر ها، عملگر های باینری هستند و اشتراک پذیری از سمت چپ دارند.

جدول 6-8. عملگر های محاسباتی ساده

|  |  |  |
| --- | --- | --- |
| عملگر | نام | شرح |
| + | جمع | دو عملوند را با یکدیگر جمع می کند. |
| - | تفریق | دو عملوند را از یکدیگر کم می کند. |
| \* | ضرب | دو عملوند را با یکدیگر ضرب می کند. |
| / | تقسیم | اولین عملوند را بر دومین عملوند تقسیم می کند. در تقسیم اعداد صحیح نتیجه تقسیم گرد می شود. |

عملگر های محاسباتی، عملیات های محاسباتی استاندارد را برای تمام نوع های از پیش تعریف شده انجام می دهد.

در زیر نمونه هایی از عملگر های ریاضی وجود دارد:

|  |
| --- |
| int x1 = 5 + 6;  int x2 = 12 - 3;  int x3 = 3 \* 4;  int x4 = 10 / 3;    double d1 = 5.0 + 6.0;  double d2 = 12.0 - 3.0;  double d3 = 3.0 \* 4.0;  double d4 = 10.0 / 3.0;  byte b1 = 5 + 6;  sbyte sb1 = 6 \* 5; |

عملگر باقی مانده

عملگر باقی مانده[[6]](#footnote-6) (%) تقسیم عملوند اول با عملوند دوم را نادیده گرفته و خارج قسمت و باقی مانده را برمی گرداند. جدول 7-8 این عملگر را شرح داده است.

عملگر باقی مانده، عملگری باینری است و شرکت پذیری از سمت چپ دارد.

جدول 7-8. عملگر باقی مانده

|  |  |  |
| --- | --- | --- |
| عملگر | نام | شرح |
| % | باقی مانده | عملوند اول را بر عملوند دوم تقسیم می کند و باقی مانده را برمی گرداند. |

خطوط زیر نمونه هایی از عملگر باقی مانده اعداد صحیح را نشان می دهد:

* 0 % 3 = 0، زیرا تقسیم 0 بر 3 برابر 0 است و باقی مانده آن 0 خواهد شد.
* 1 % 3 = 1، زیرا تقسیم 1 بر 3 برابر 0 است و باقی مانده آن 1 خواهد شد.
* 2 % 3 = 2، زیرا تقسیم 2 بر 3 برابر 0 است و باقی مانده آن 2 خواهد شد.
* 3 % 3 = 0، زیرا تقسیم 3 بر 3 برابر 1 است و باقی مانده آن 0 خواهد شد.
* 4 % 3 = 1، زیرا تقسیم 4 بر 3 برابر 1 است و باقی مانده آن 1 خواهد شد.

همچنین عملگر باقی مانده می تواند برای بدست آوردن باقی مانده واقعی با اعداد واقعی مورد استفاده قرار گیرد.

|  |
| --- |
| Console.WriteLine("0.0f % 1.5f is {0}" , 0.0f % 1.5f);  Console.WriteLine("0.5f % 1.5f is {0}" , 0.5f % 1.5f);  Console.WriteLine("1.0f % 1.5f is {0}" , 1.0f % 1.5f);  Console.WriteLine("1.5f % 1.5f is {0}" , 1.5f % 1.5f);  Console.WriteLine("2.0f % 1.5f is {0}" , 2.0f % 1.5f);  Console.WriteLine("2.5f % 1.5f is {0}" , 2.5f % 1.5f); |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| 0.0f % 1.5f is 0 // 0 باقی مانده 0.0 / 1.5 = 0  0.5f % 1.5f is 0.5 // .5باقی مانده 0.5 / 1.5 = 0  1.0f % 1.5f is 1 // 1 باقی مانده 1.0 / 1.5 = 0  1.5f % 1.5f is 0 // 0 باقی مانده 1.5 / 1.5 = 1  2.0f % 1.5f is 0.5 // .5باقی مانده 2.0 / 1.5 = 1  2.5f % 1.5f is 1 // 1 باقی مانده 2.5 / 1.5 = 1 |

عملگر های رابطه ای و تساوی

عملگر های رابطه ای و تساوی، عملگر هایی باینری هستند که عملوند های خود را مقایسه می کند و مقداری از نوع بولین را بر می گردانند. جدول 8-8 این عملگر ها را لیست کرده است.

عملگر های رابطه ای و تساوی، عملگر هایی باینری هستند و شرکت پذیری از سمت چپ دارند.

جدول 8-8. عملگر های رابطه ای و تساوی

|  |  |  |
| --- | --- | --- |
| عملگر | نام | شرح |
| < | کوچکتر | اگر عملوند اول از عملوند دوم کوچکتر باشد مقدار true و در غیر این صورت مقدار false را برمی گرداند. |
| > | بزرگتر | اگر عملوند اول از عملوند دوم بزرگتر باشد مقدار true و در غیر این صورت مقدار false را برمی گرداند. |
| <= | کوچکتر یا مساوی | اگر عملوند اول از عملوند دوم کوچکتر یا مساوی باشد مقدار true و در غیر این صورت مقدار false را برمی گرداند. |
| >= | بزرگتر یا مساوی | اگر عملوند اول از عملوند دوم بزرگتر یا مساوی باشد مقدار true و در غیر این صورت مقدار false را برمی گرداند. |
| == | مساوی | اگر عملوند اول با عملوند دوم مساوی باشد مقدار true و در غیر این صورت مقدار false را برمی گرداند. |
| != | نا مساوی | اگر عملوند اول با عملوند دوم مساوی باشد مقدار false و در غیر این صورت مقدار true را برمی گرداند. |

یک عبارت باینری با عملگر مساوی یا رابطه ای، یک مقدار از نوع بولین برمی گرداند.

|  |
| --- |
| * توجه برخلاف C و C++، در C# اعداد تفسیری از بولین ندارند. |

|  |
| --- |
| int x = 5;  if( x ) // است، و از نوع بولین نیست. int از نوع xاشتباه.  ...  if( x == 5 ) // خوب، زیرا این عبارت مقداری از نوع بولین را برمیگرداند.  ... |

هنگامی که مقادیر بولین چاپ می شوند، در خروجی رشته false یا true نمایش داده می شود.

|  |
| --- |
| int x = 5, y = 4;  Console.WriteLine("x == x is {0}" , x == x);  Console.WriteLine("x == y is {0}" , x == y); |

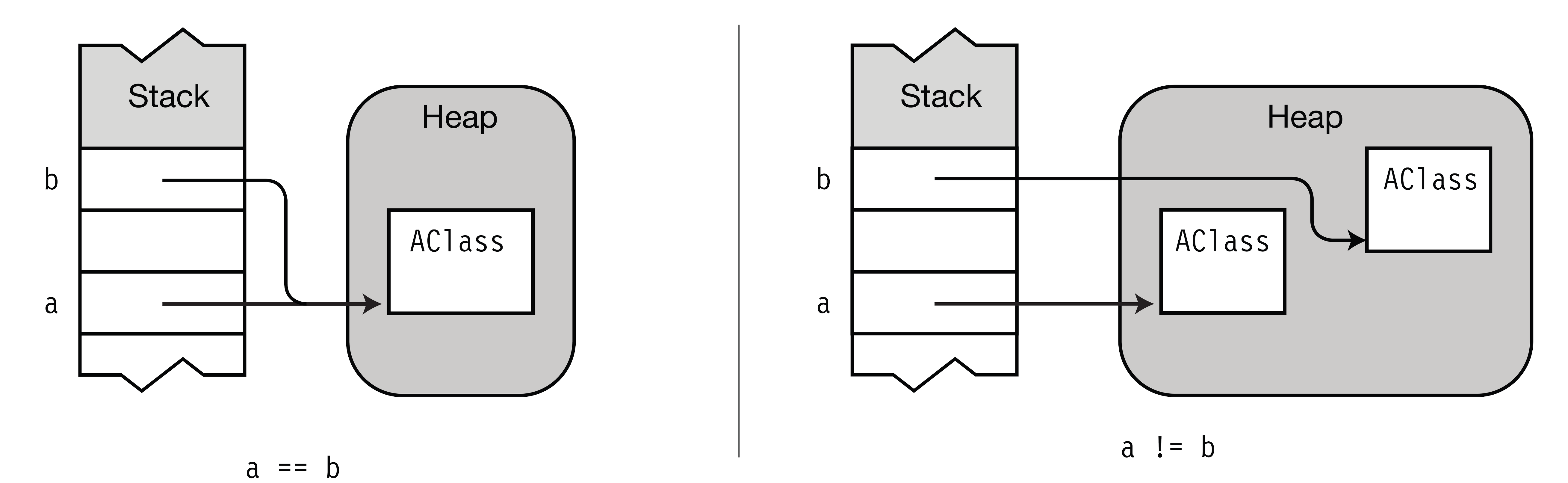
این کد خروجی زیر را تولید می کند:

|  |
| --- |
| x == x is True  x == y is False |

عملگر های تساوی

هنگام که نوع های ارجاعی با عملگر تساوی مقایسه می شوند، تنها مرجع آنها مورد مقایسه قرار داده می شود.

* اگر مرجع های یکسان باشند، به این معنا که اگر آنها به شی یکسانی در حافظه اشاره کنند، این مقایسه مقدار true را برمی گرداند، در غیر این صورت حتی اگر دو شی در حافظه معادل یکدیگر هم باشند مقدار false را برمی گرداند.
* این مقایسه، مقایسه کم عمق[[7]](#footnote-7) نامیده می شود.
* شکل 4-8 مقایسه نوع های ارجاعی را نشان می دهد.
* در سمت چپ شکل، مرجع نگه داری شده توسط A و B یکسان هستند، بنابراین مقایسه این دو مقدار true را برمی گرداند.
* در سمت راست شکل، مرجع ها یکسان نیستند، بنابراین حتی اگر محتویات هر دو شی دقیقا یکسان باشند، این مقایسه مقدار false را برمی گرداند.



شکل 4-8. مقایسه نوع های ارجاعی با عملگر تساوی

شی های نوع string نوع های ارجاعی هستند اما مقایسه آنها متفاوت است. هنگامی که رشته ها با عملگر های تساوی مقایسه می شوند، آنها نسبت به تعداد حروف و کوچک و بزرگ بودن حروف مقایسه می شوند.

* اگر دو رشته طول و حروف کوچک و بزرگ یکسانی داشته باشند، مقایسه تساوی مقدار true برمی گرداند، حتی اگر آنها در مناطق مختلفی از حافظه ذخیره شده باشند.
* این مقایسه، مقایسه عمیق[[8]](#footnote-8) نامیده می شود.

Delegate ها[[9]](#footnote-9) که در فصل 15 بررسی می شوند، نوع های ارجاعی هستند و همچنین برای مقایسه آنها از مقایسه عمیق استفاده می شود. هنگامی که delegate ها با عملگر تساوی مقایسه می شوند، اگر هر دو نماینده null باشند یا هر دو نماینده تعداد عضوها یکسانی در لیست فراخوانی های خود داشته باشند، مقایسه مقدار true را برمی گرداند.

هنگام مقایسه عبارات عددی، نوع ها و مقادیر مقایسه می شوند. هنگام مقایسه نوع های شمارشی[[10]](#footnote-10)، مقادیر اساسی عملوند ها مقایسه می شوند. نوع های شمارشی را در فصل 13 بررسی خواهیم کرد.

عملگر های افزایش و کاهش

عملگر افزایش یک واحد به عملوند می اضافه می کند. عملگر کاهش یک واحد از عملوند خود کم می کند. جدول 9-8 عملگر ها را لیست کرده است و آنها را شرح می دهد.

این عملگر ها، عملگر های یگانی هستند. و دو صورت پیشوندی و پسوندی آنها متفاوت عمل می کنند.

* در پیشوندی، عملگر قبل از عملوند قرار می گیرد. به عنوان مثال، ++x و --y
* در پسوندی، عملگر بعد از عملوند قرار می گیرد. به عنوان مثال، x++ و y--

جدول 9-8. عملگر های افزایش و کاهش

|  |  |  |
| --- | --- | --- |
| عملگر | نام | شرح |
| ++ | پیش افزایش ++var | ابتدا مقدار متغییر یک واحد افزایش می یابد و سپس از این متغییر استفاده می کند. |
|  | پس افزایش var++ | ابتدا از متغییر استفاده می کند و سپس مقدار متغییر یک واحد افزایش می یابد. |
| -- | پیش کاهش --var | ابتدا مقدار متغییر یک واحد کاهش می یابد و سپس از این متغییر استفاده می کند. |
|  | پس کاهش var-- | ابتدا از متغییر استفاده می کند و سپس مقدار متغییر یک واحد کاهش می یابد. |

در مقایسه عملگر های پیشوندی و پسوندی

* در پایان، صرف نظر از اینکه آیا عملگر قبل یا بعد از عملوند استفاده می شود، مقدار در متغییر عملوند بعد از مقدار یکسانی ذخیره می شود.
* تنها تفاوت در مقدار برگشت داده شده عملگر به عبارت است.

جدول 10-8 خلاصه ای از رفتار این عملگر ها را نشان می دهد.

جدول 10-8. رفتار پیشوند و پسوندی عملگر های افزایش و کاهش

|  |  |  |  |
| --- | --- | --- | --- |
|  | شروع عبارت x=10 | مقدار برگشتی به عبارت | مقدار متغییر بعد از ارزیابی |
| پیش افزایش | ++x | 11 | 11 |
| پس افزایش | x++ | 10 | 11 |
| پیش کاهش | --x | 9 | 9 |
| پس کاهش | x-- | 10 | 9 |

به عنوان مثال، در زیر این چهار عملگر نشان داده شده است. برای نشان دادن نتایج مختلف هر عملگر، قبل از استفاده از این عملگر ها متغییر x را با مقدار 5 مقدار دهی کرده ایم.

|  |
| --- |
| int x = 5, y;  y = x++; // y: 5, x: 6 نتیجه:  Console.WriteLine("y: {0}, x: {1}" , y, x);  x = 5;  y = ++x; // y: 6, x: 6 نتیجه:  Console.WriteLine("y: {0}, x: {1}" , y, x);  x = 5;  y = x--; // y: 5, x: 4 نتیجه:  Console.WriteLine("y: {0}, x: {1}" , y, x);  x = 5;  y = --x; // y: 4, x: 4 نتیجه:  Console.WriteLine("y: {0}, x: {1}" , y, x); |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| y: 5, x: 6  y: 6, x: 6  y: 5, x: 4  y: 4, x: 4 |

عملگر های منطقی شرطی

عملگر های منطقی برای مقایسه یا نقض ارزشهای منطقی عملوند ها استفاده می شوند و نتیجه عملگر استفاده شده را باز میگرداند. جدول 11-8 این عملگر ها را لیست کرده است.

عملگر های AND و OR منطقی، عملگر هایی باینری هستند و شرکت پذیری از سمت چپ دارند. عملگر NOT منطقی، عملگر یگانی است.

جدول 11-8. عملگر های منطقی شرطی

|  |  |  |
| --- | --- | --- |
| عملگر | نام | شرح |
| && | AND منطقی | اگر هر دو عملوند true باشند، true برمی گرداند در غیر این صورت false برمیگرداند. |
| || | OR منطقی | اگر یکی از عملوند ها true باشد، true برمی گرداند در غیر این صورت false برمی گرداند. |
| ! | NOT منطقی | اگر عملوند false باشد، true برمی گرداند در غیر این صورت false بر می گرداند. |

گرامر این عملگر ها به شرح زیر است، که در آن Expr1 و Expr2 مقادیری از نوع bool هستند.

|  |
| --- |
| Expr1 && Expr2  *Expr1* || *Expr2*  ! *Expr* |

در زیر نمونه هایی از این عملگر ها وجود دارد:

|  |
| --- |
| bool bVal;  bVal = (1 == 1) && (2 == 2); // True, هستندtrueهر دو عملوند  bVal = (1 == 1) && (1 == 2); // False, استfalse عملوند دوم  bVal = (1 == 1) || (2 == 2); // True, هستندtrueهر دو عملوند  bVal = (1 == 1) || (1 == 2); // True, استtrueعملوند اول  bVal = (1 == 2) || (2 == 3); // False, هستندfalseهر دو عملوند  bVal = true; // تعیین می شودbVal برای متغییر true مقدار  bVal = !bVal; // تعیین می شودbValبرای متغییر false حالا مقدار |

عملگر های منطقی شرطی به صورت اتصال کوتاه عمل می کنند، به این معنا که، اگر بعد از ارزیابی Expr1 نتیجه را بتواند تعیین کند پس دیگر Expr2 را ارزیابی نخواهد کرد. کد زیر نمونه ای از این عبارت را نشان می دهد که در آن مقدار بعد از ارزیابی عملوند اول می تواند تعیین شود.

|  |
| --- |
| bool bVal;  bVal = (1 == 2) && (2 == 2); // ، بعد از ارزیابی اولین عبارتFalse  bVal = (1 == 1) || (1 == 2); // ، بعد از ارزیابی اولین عبارتTrue |

بخاطر رفتار اتصال کوتاه، عبارات را با اثرات جانبی (مثل تغییر دادن یک مقدار) در Expr2 قرار ندهید، زیرا آنها ممکن است ارزیابی نشوند. در کد زیر، دستور افزایش متغییر iVal اجرا نخواهد شد، زیرا بعد از اجرا دستور اول، می تواند ارزیابی کند که مقدار کل دستور false است.

|  |
| --- |
| bool bVal; int iVal = 10;  bVal = (1 == 2) && (9 == iVal++); // bVal = False, iVal = 10نتیجه :  ↑ ↑  False هرگز ارزیابی نمی شود |

عملگر های منطقی

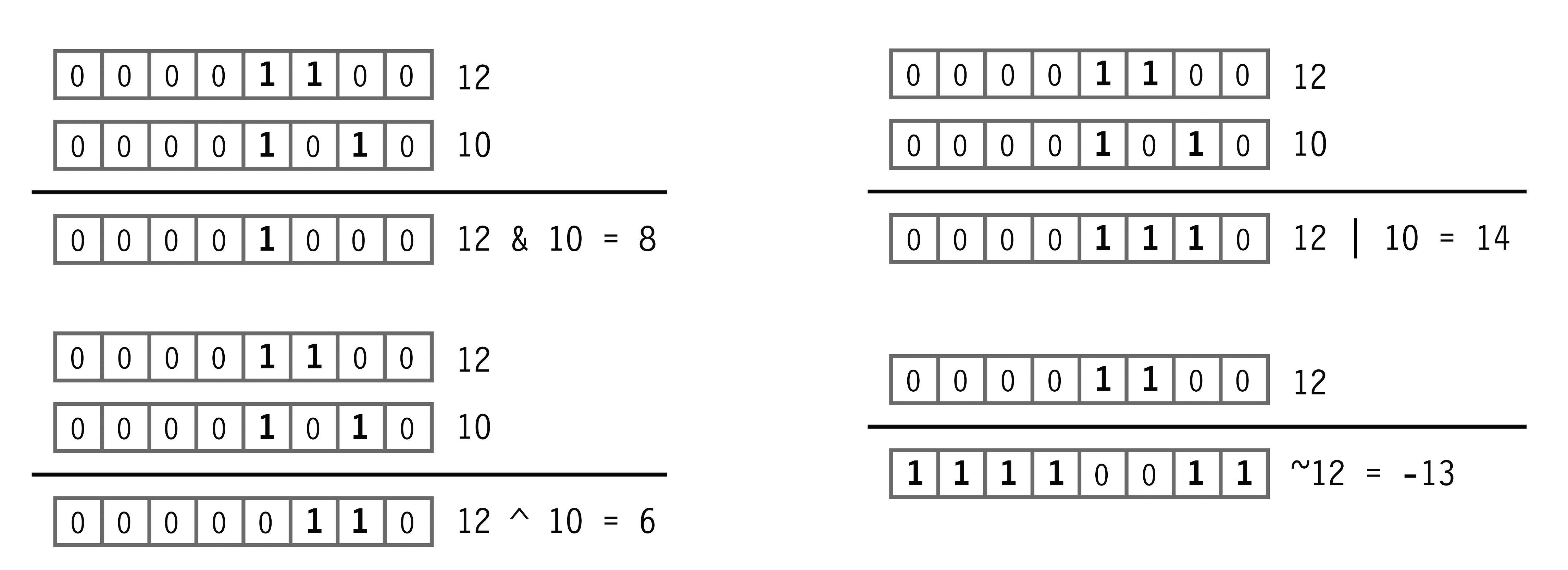
عملگر های منطقی بیتی اغلب برای تعیین الگوهای بیتی در پارامتر متدها استفاده می شود. جدول 12-8 عملگر های منطقی بیتی را لیست کرده است.

جدول 12-8. عملگر های منطقی

|  |  |  |
| --- | --- | --- |
| عملگر | نام | شرح |
| & | AND بیتی | AND بیتی از دو عملوند تولید می شود. اگر بیت هر دو عملوند 1 باشد بیت نتیجه 1 است. |
| | | OR بیتی | OR بیتی از دو عملوند تولید می شود. اگر بیت هر یک از عملوند ها 1 باشد بیت نتیجه 1 است. |
| ^ | XOR بیتی | XOR بیتی از دو عملوند تولید می شود. اگر تنها بیت یکی از عملوند ها 1 باشد بیت نتیجه 1 است. |
| ~ | نقیض بیتی | مخالف بیت عملوند آن انتخاب می شود. یک عملوند مکمل تولید می کند. ( یک مکمل مخالف هر یک از بیت های آن را بر می گرداند، اگر 0 باشد آن را به 1 و اگر 1 باشد آن را به 0 تغییر می دهد. |

عملگر های بیتی باینری، بیت های متناظر در هر یک از عملوند ها را مقایسه می کند و آنها را با توجه به عملیات منطقی در مقدار بازگشتی تعیین می کند.

شکل 5-8 چهار نمونه از عملیات های منطقی بیتی را نشان می دهد.



شکل 5-8. مونه هایی از عملگر های منطقی بیتی

کد زیر نمونه های قبلی را پیاده سازی می کند:

|  |
| --- |
| const byte x = 12, y = 10;  sbyte a;  a = x & y; // a = 8  a = x | y; // a = 14  a = x ^ y; // a = 6  a = ~x; // a = -13 |

عملگر های شیفت

عملگر های شیفت بیتی الگوی بیت را با تعداد مشخصی به سمت راست یا چپ انتقال می دهد. جدول 13-8 عملگر های شیفت را لیست کرده است.

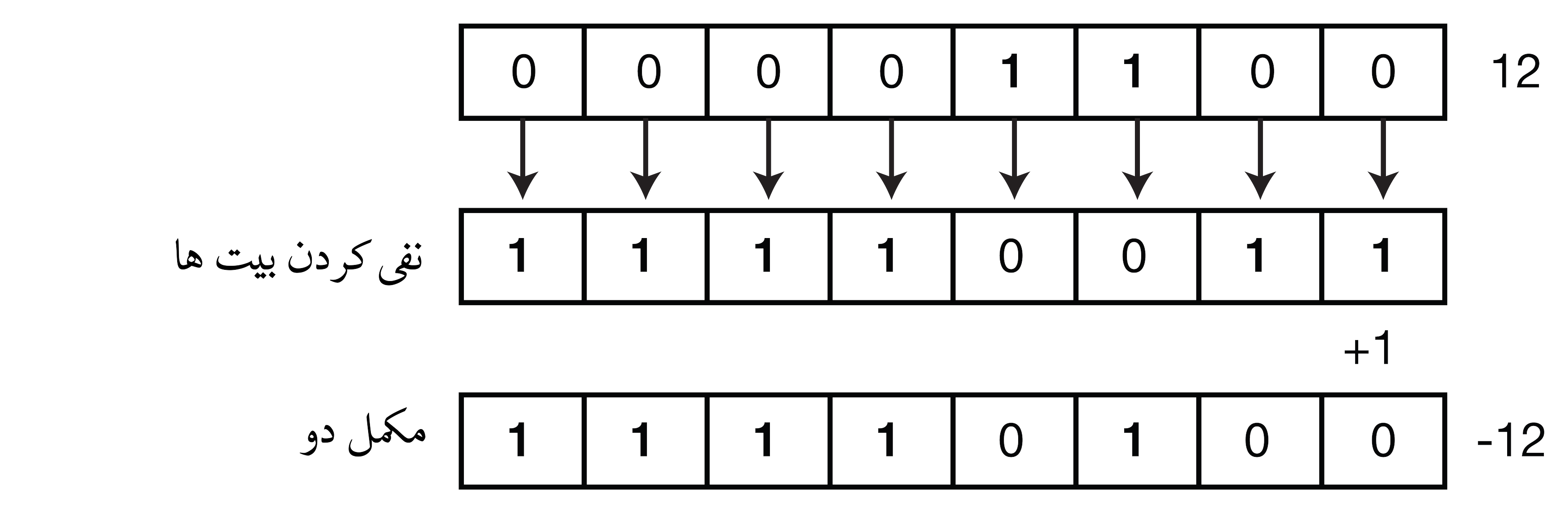
عملگر های شیفت، عملگر هایی باینری هستند و شرکت پذیری از سمت چپ دارند. گرامر عملگر های شیفت بیتی در زیر نشان داده شده است. تعداد موقعیت مورد انتقال با Count نشان داده شده است.

|  |
| --- |
| *Operand* << *Count* // شیفت به چپ  *Operand* >> *Count* // شیفت به راست |

جدول 13-8. عملگر های شیفت

|  |  |  |
| --- | --- | --- |
| عملگر | نام | شرح |
| << | شیفت به چپ | الگوی بیت به سمت چپ موقعیت، نسبت به تعداد داده شده انتقال می یابد. بیت های انتقال داده شده به انتهای سمت چپ از دست داده می شوند. موقعیت های ایجاد شده در سمت راست با 0 پر می شوند |
| >> | شیفت به راست | الگوی بیت به سمت راست موقعیت، نسبت به تعداد داده شده انتقال می یابد. بیت های انتقال داده شده به انتهای سمت راست از دست داده می شوند. |

برای بیشتر برنامه های C# نیازی نیست که همه چیز را در مورد سطح پایین سخت افزار بدانید. با توجه به اینکه اگر بخواهید بیت اعداد را دستکاری کنید، دانستن اینکه اعداد در سطح پایین سخت افزار چگونه نمایش داده می شود، می تواند مفید باشد. سطح پایین سخت افزار اعداد باینری را به صورت مکمل دو نمایش می دهد. در نمایش مکمل دو، اعداد مثبت فرم باینری عادی خود را دارند. برای منفی کردن یک عدد، باید نقیض تک تک بیت های آن عدد را با عدد یک جمع کنید. این فرایند یک عدد مثبت را به منفی و یک عدد منفی را به مثبت تبدیل می کند. در مکمل دو سمت چپ بیتی تمام اعداد منفی دارای عدد یک است. شکل 6-8 عدد منفی 12 را نمایش می دهد.



شکل 6-8. برای تولید عدد منفی با استفاده از روش مکمل دو، تمام بیت ها را نفی کرده و به آن یک واحد اضافه می کنیم.

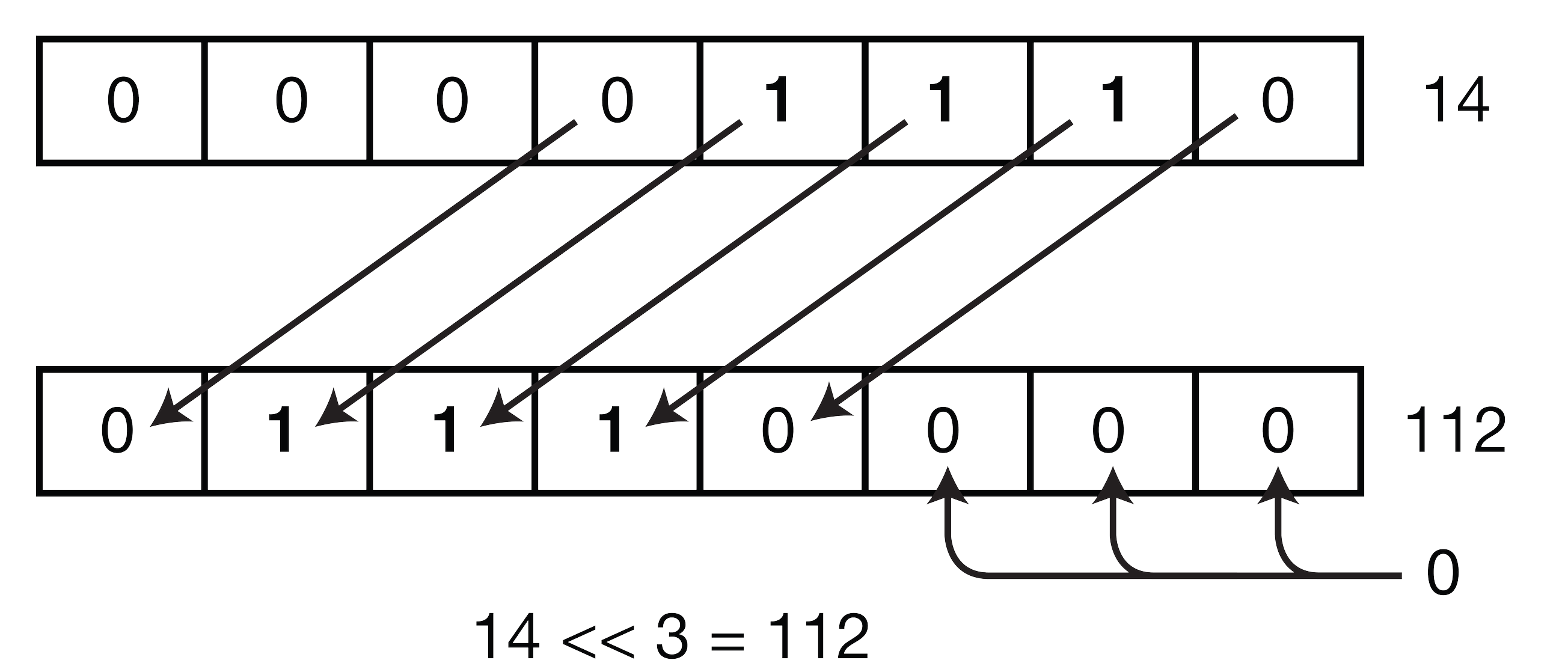
هنگامی که یک عدد شیفت داده می شود نمایش باینری آن مهم است، زیرا نتیجه شیفت کردن یک مقدار صحیح به سمت چپ همانند ضرب آن با عدد دو است. و شیفت کردن آن عدد به سمت راست همانند تقسیم آن به عدد دو است.

با این حال اگر یک عدد منفی به سمت راست شیفت داده شود سمت چپ ترین بیت با صفر پر می شود، که این عمل باعث تولید نتیجه اشتباه می شود. صفر در سمت راست، یک عدد مثبت را نشان می دهد. که این اشتباه است زیرا تقسیم یک عدد منفی به 2، یک عدد مثبت تولید نمی کند.

برای حل این موضوع، زمانی که عملوند یک عدد صحیح علامت دار است، اگر سمت چپ ترین بیت عملوند 1 (نشان دهنده یک عدد منفی) باشد، بیت های سمت چپ به جای 0 با 1 پر می شوند. این کار نمایش صحیح مکمل دو را حفظ می کند. برای اعداد صحیح یا بدون علامت، بیت های سمت چپ با 0 پر می شوند.

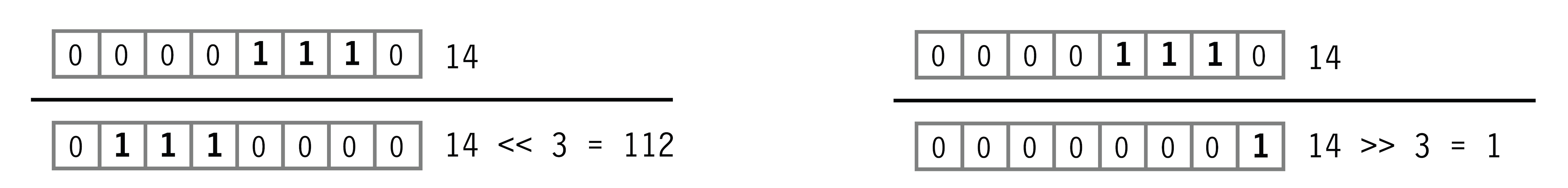
شکل 7-8 چگونگی ارزیابی دستور 14<<3 را در یک بایت نشان می دهد. این عمل به شرح زیر است:

* هر یک از بیت های عملوند (14) سه مکان به سمت چپ منتقل می شوند.
* سه مکان خالی سمت راست با صفر پر می شوند.
* مقدار بدست آمده 112 است.



شکل 7-8. مثالی از شیفت دادن به سمت چپ

شکل 8-8 عملگر های شیفت بیت ها را نشان می دهد.



شکل 8-8. شیفت بیت ها

کد زیر نمونه های قبلی را پیاده سازی می کند:

|  |
| --- |
| int a, b, x = 14;  a = x << 3; // شیفت به سمت چپ  b = x >> 3; // شیفت به سمت راست  Console.WriteLine("{0} << 3 = {1}" , x, a);  Console.WriteLine("{0} >> 3 = {1}" , x, b); |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| 14 << 3 = 112  14 >> 3 = 1 |

عملگر های انتساب

عملگر های انتساب بعد از ارزیابی عبارت سمت راست، با استفاده از آن مقدار متغییر سمت چپ عملگر را تعیین می کند. جدول 14-8 عملگر های انتساب را لیست کرده است.

عملگر های انتساب، عملگر هایی باینری و شرکت پذیری از سمت راست دارند.

جدول 14-8. عملگر های انتساب

|  |  |
| --- | --- |
| عملگر | شرح |
| = | عملگر انتساب ساده، دستور سمت راست عملگر را ارزیابی می کند، و مقدار برگردانده شده را به متغییر سمت چپ عملگر اختصاص می دهد. |
| \*= | انتساب ترکیبی؛ var \*= expr برابر با var = var \* (expr) . |
| /= | انتساب ترکیبی؛ var /= expr برابر با var = var / (expr) . |
| %= | انتساب ترکیبی؛ var %= expr برابر با var = var % (expr) . |
| += | انتساب ترکیبی؛ var += expr برابر با var = var + (expr) . |
| -= | انتساب ترکیبی؛ var -= expr برابر با var = var - (expr) . |
| <<= | انتساب ترکیبی؛ var <<= expr برابر با var = var << (expr) . |
| >>= | انتساب ترکیبی؛ var >>= expr برابر با var = var >> (expr) . |
| &= | انتساب ترکیبی؛ var &= expr برابر با var = var & (expr) . |
| ^= | انتساب ترکیبی؛ var ^= expr برابر با var = var ^ (expr) . |
| |= | انتساب ترکیبی؛ var |= expr برابر با var = var | (expr) . |

گرامر این عملگر ها به شکل زیر است:

|  |
| --- |
| متغییر عملگر عبارت |

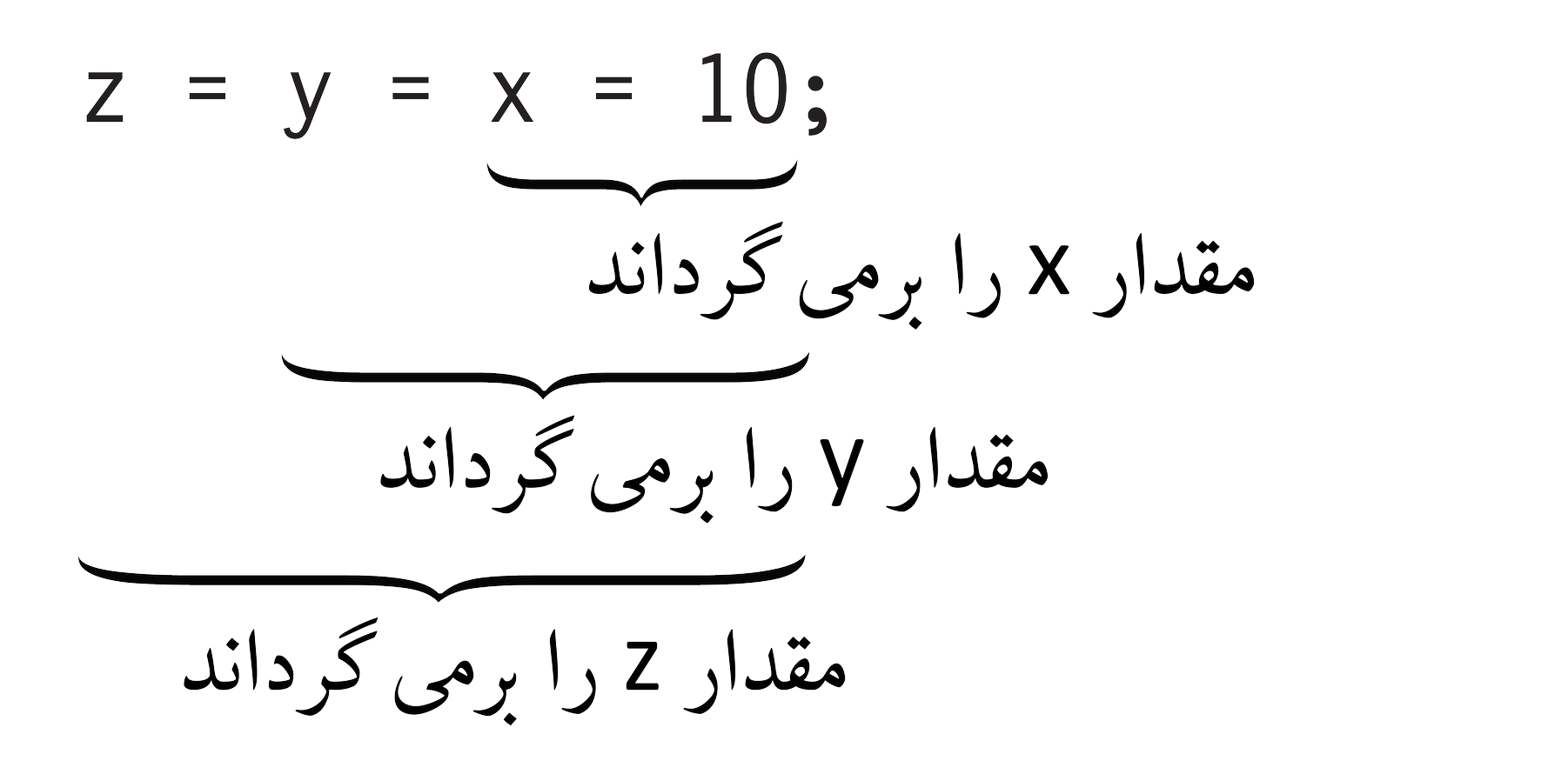
برای انتساب ساده عبارت سمت راست عملگر انتساب ارزیابی می شود و مقدار آن به متغییر سمت چپ اختصاص داده می شود.

|  |
| --- |
| int x;  x = 5;  x = y \* z; |

به یاد داشته باشید که عبارت استفاده شده در گرامر انتساب یک عبارت است، و در نتیجه مقداری را به موقعیت خود برمی گرداند. بعد از اینکه انتساب انجام شود، مقدار عبارت انتساب، مقدار عملوند سمت چپ است. بنابراین، در این مورد عبارت x=10 ، مقدار 10 به متغییر x اختصاص داده می شود. در حال حاضر مقدار x برابر 10 برابر با مقدار کل عبارت می شود.

همانطور که در شکل 9-8 نشان داده شده است. بدلیل اینکه انتساب یک عبارت است، پس آن می تواند بخشی از یک عبارت بزرگتر باشد، ارزیابی عبارت به شرح زیر است:

* از آنجایی که انتساب شرکت پذیری از سمت راست دارد، ارزیابی از سمت راست شروع می شود و مقدار 10 به متغییر x اختصاص داده می شود.
* که این عبارت، عملوند سمت راست عملگر انتساب برای متغییر y است، به طوری که مقدار x که در حال حاضر 10 است به متغییر y اختصاص داده می شود.
* انتساب متغییر y، عملوند سمت راست عملگر انتساب برای متغییر z است. هر سه متغییر با مقدار 10 مقدار دهی می شوند.



شکل 9-8. یک عبارت انتساب بعد از عمل انتساب، مقدار سمت چپ عملگر خود را برمی گرداند.

انواعی از اشیا که می توانند در سمت راست عملگر انتساب باشد به شرح زیر است.

* متغییر ها (متغیرهای محلی، فیلد ها، پارامتر ها)
* خصوصیت ها
* ایندکسر ها
* رویداد ها

انتساب ترکیبی

در اغلب اوقات، یک عبارت ارزیابی می شود و مقدار آن به متغییر اضافه می شود، همانطور که در زیر نشان داده شده است:

|  |
| --- |
| x = x + expr; |

عملگر های انتساب ترکیبی اجازه می دهند که از تکرار متغییر سمت چپ در سمت راست تحت شرایط خاص جلوگیری شود. به عنوان مثال، دو دستور زیر معنایی معادل یکدیگر دارند، اما دومی کوتاه تر است.

|  |
| --- |
| x = x + (y – z);  x += y – z; |

دیگر دستورات انتساب ترکیبی مشابه این هستند:

|  |
| --- |
| به پرانتزها توجه کنید.  ↓ ↓  x \*= y – z; // معادل x = x \* (y – z)  x /= y – z; // معادل x = x / (y – z)  ... |

عملگر شرطی

عملگر شرطی یک راه قدرتمند و مختصر است که مقداری را از میان دو مقدار بر اساس شرط خاصی بر می گرداند. جدول 15-8 این عملگر را نشان می دهد.

جدول 15-8. عملگر انتساب شرطی

|  |  |  |
| --- | --- | --- |
| عملگر | نام | شرح |
| ?: | عملگر شرطی | یک عبارت را ارزیابی می کند و بسته به اینکه آیا دستور شرط true یا false است یکی از دو مقدار را برمی گرداند. |

گرامر این عملگر به شرح زیر است.

* شرط باید مقداری از نوع بولین را برگرداند.
* اگر شرط true باشد، Experssion1 ارزیابی می شود و مقدار آن را برمی گرداند، در غیر این صورت Experssion2 ارزیابی می شود و مقدار آن برگشت داده می شود.

|  |
| --- |
| شرط ? Expression1 : Expression2 |

عملگر شرطی را می توان با if…else نیز ایجاد کرد. به عنوان مثال، در زیر if…else یک شرط را چک می کند، و اگر شرط درست باشد مقدار 5 به متغییر intVar اختصاص داده می شود. در غیر این صورت مقدار 10 به آن اختصاص داده می شود.

|  |
| --- |
| if ( x < y ) // if...else  intVar = 5;  else  intVar = 10; |

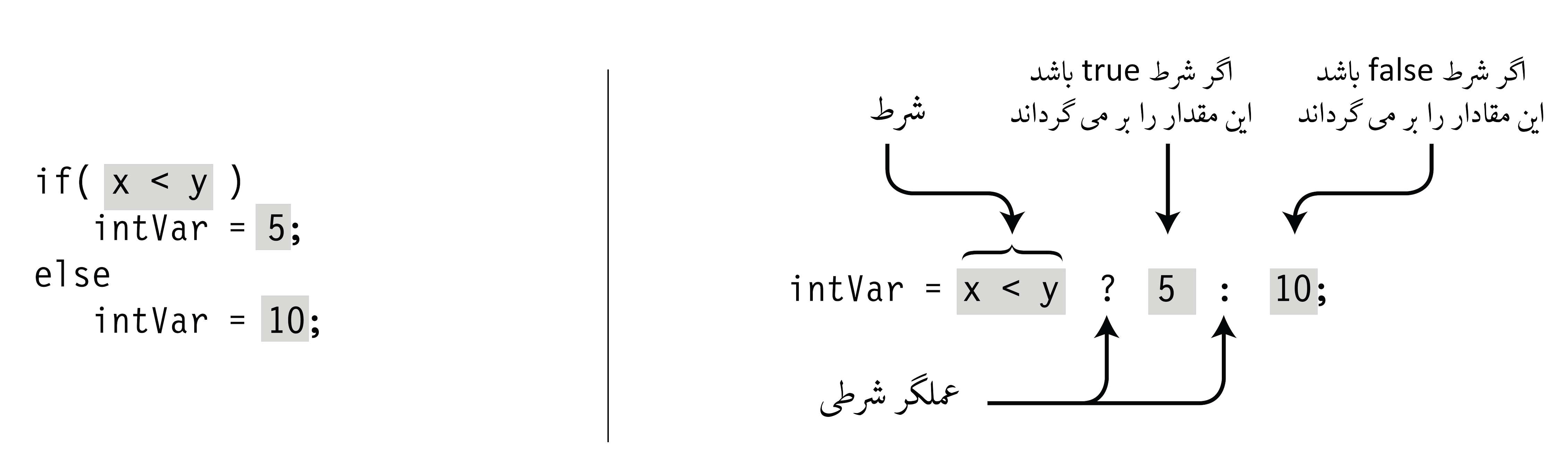
همانطور که در دستور زیر نشان داده شده است، عملگر شرطی می تواند همان عمل را به طور مختصر انجام دهد:

|  |
| --- |
| intVar = x < y ? 5 : 10; // عملگر شرطی |

در کد زیر، به قرار دادن شرط و هر یک از عبارات در خطوط جداگانه، باعث می شود که درک بهتری از آن داشته باشید.

|  |
| --- |
| intVar = x < y  ? 5  : 10; |

شکل 10-8 مقایسه این دو دستور را نشان می دهد.



شکل 10-8. مقایسه عملگر شرطی با if…else

به عنوان مثال، کد زیر سه بار از عملگر شرطی در هر یک از عبارت WriteLine استفاده می کند. و اولین مورد، مقدار x یا y را برمی گرداند. دو مورد بعدی، یک رشته خالی یا رشته "not" را برمی گرداند.

|  |
| --- |
| int x = 10, y = 9;  int highVal = x > y // شرط  ? x // Expression 1  : y; // Expression 2  Console.WriteLine("highVal: {0}\n" , highVal);  Console.WriteLine("x is{0} greater than y" ,  x > y // شرط  ? "" // Expression 1  : " not" ); // Expression 2  y = 11;  Console.WriteLine("x is{0} greater than y" ,  x > y // شرط  ? "" // Expression 1  : " not" ); // Expression 2 |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| highVal: 10  x is greater than y  x is not greater than y |

|  |
| --- |
| * توجه دستور if…else یک عبارت کنترلی است. و یکی از دو عمل را انجام می دهد. عملگر شرطی برای برگرداندن مقداری از میان دو مقدار استفاده می شود. |

عملگر های یگانی ریاضی

عملگر های یگانی علامت یک مقدار عددی را تعیین می کند. آنها در جدول 16-8 لیست شده اند.

* عملگر یگانی مثبت به سادگی مقدار عملوند را برمی گرداند.
* عملگر یگانی منفی، مقدار عملوند را بر 0 تفریق می کند و آن را برمی گرداند.

جدول 16-8. عملگر های یگانی

|  |  |  |
| --- | --- | --- |
| عملگر | نام | شرح |
| + | علامت مثبت | مقدار عددی عملوند را برمی گرداند |
| - | علامت منفی | مقدار عددی عملوند را از 0 کم می کند و آن را برمی گرداند |

به عنوان مثال، کد زیر استفاده و نتیجه این عملگر ها را نشان می دهد:

|  |
| --- |
| int x = +10; // x = 10  int y = -x; // y = -10  int z = -y; // z = 10 |

تبدیل نوع های تعریف شده توسط کاربر

تبدیل نوع های تعریف شده توسط کاربر در جزئیات بیشتری در فصل 16 بررسی شده اند، اما به دلیل اینکه آنها عملگر هستند، آنها را در اینجا ذکر می کنیم.

* می توان هر دو تبدیل ضمنی و صریح را برای کلاس ها و ساختارها تعریف کرد. این اجازه می دهد تا تعریف شده توسط کاربر به برخی از نوع های دیگر و بالعکس تبدیل شود.
* C# تبدیل ضمنی و صریح را فراهم می کند.
  + با تبدیل ضمنی، اگر لازم باشد کامپایلر به طور خودکار این تبدیل را انجام می دهد، زمانی آن را انجام می دهد که در مکانی خاص باید از آن نوع استفاده شود.
  + با تبدیل صریح، کامپایلر تنها زمانی که از عملگر استفاده می شود این تبدیل را انجام می دهد.

گرامر تبدیل ضمنی به شرح زیر است. اصلاحگر های public و static برای تمام تبدیل های تعریف شده توسط کاربر مورد نیاز است.

|  |
| --- |
| ضروری نوع مقصد نوع مبداء  ↓   ↓ ↓  public static implicit operator *TargetType* ( *SourceType Identifier* )  {  ...  return *ObjectOfTargetType*;  } |

گرامر تبدیل صریح نیز به همان صورت است، اما به جای implicit ، explicit جایگزین آن می شود.

کد زیر نمونه از دستورات را برای عملگر تبدیل نشان می دهد که نوع مقصد LimitedInt را به نوع مبداء int و بالعکس تبدیل می کند.

|  |
| --- |
| class LimitedInt مقصد مبداء  { ↓ ↓  public static implicit operator int (LimitedInt li) // int به LimitedInt  {  return li.TheValue;  } مقصد مبداء  ↓ ↓  public static implicit operator LimitedInt (int x) // LimitedInt به int  {  LimitedInt li = new LimitedInt();  li.TheValue = x;  return li;  }  private int \_theValue = 0;  public int TheValue{ ... }  } |

به عنوان مثال، کد زیر از دو عملگر برای تبدیل دو نوع مختلف استفاده می کند. در متد Main، لیترالی از نوع int به شی ای از نوع LimitedInt تبدیل می شود، و در خط بعدی یک شی LimitedInt به int تبدیل می شود.

|  |
| --- |
| class LimitedInt  {  const int MaxValue = 100;  const int MinValue = 0;  public static implicit operator int(LimitedInt li) // تبدیل نوع  {  return li.TheValue;  }  public static implicit operator LimitedInt(int x) // تبدیل نوع  {  LimitedInt li = new LimitedInt();  li.TheValue = x;  return li;  }  private int \_theValue = 0;  public int TheValue // خصوصیت  {  get { return \_theValue; }  set  {  if (value < MinValue)  \_theValue = 0;  else  \_theValue = value > MaxValue  ? MaxValue  : value;  }  }  }  class Program  {  static void Main() // Main  {  LimitedInt li = 500; // LimitedInt به 500تبدیل  int value = li; // int به LimitedIntتبدیل  Console.WriteLine("li: {0}, value: {1}" , li.TheValue, value);  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| li: 100, value: 100 |

تبدیل صریح و عملگر کست

کد مثال قبل تبدیل ضمنی int به LimitedInt و تبدیل ضمنی LimitedInt به int را نشان داد. با این حال، اگر دو عملگر تبدیل صریح تعریف کنید، در هنگام تبدیل باید به صراحت از این عملگر ها استفاده کنید.

عملگر کست شامل نام نوع مقصد است که در داخل پرانتز قرار می گیرد. برای مثال، در کد زیر، متد Main به طور صریح مقدار 500 را به یک شی LimitedInt تبدیل کرده است.

|  |
| --- |
| عملگر کست  ↓  LimitedInt li = (LimitedInt) 500; |

به عنوان مثال، در کد زیر بخش های تغییر یافته مشخص شده است:

|  |
| --- |
| ↓  public static explicit operator int(LimitedInt li)  {  return li.TheValue;  }  ↓  public static explicit operator LimitedInt(int x)  {  LimitedInt li = new LimitedInt();  li.TheValue = x;  return li;  }  static void Main()  { ↓  LimitedInt li = (LimitedInt) 500;  int value = (int) li;  ↑  Console.WriteLine("li: {0}, value: {1}" , li.TheValue, value);  } |

در هر دو نسخه کد، خروجی زیر را تولید می کنند:

|  |
| --- |
| li: 100, value: 100 |

دو عملگر دیگر وجود دارد که مقداری از یک نوع را گرفته و مقداری از نوع مشخص شده دیگر را برمی گرداند. این دو عملگر در پایان فصل 16 بررسی شده اند.

سربار گذاری عملگر ها

همانطور که دیده می شود، عملگر های C#، برای کار با نوع های از پیش تعریف شده ای که به عنوان عملوند استفاده می شوند، تعریف شده هستند. در نوع های تعریف شده توسط کاربر با یک عملگر ساده (بدون تعریف آنها برای این نوع ها) نمی تواند گفت که چگونه این عملوند ها را پردازش کند. سربار گذاری عملگر ها این اجازه را می دهد که عملگر های C# را برای نوع های تعریف شده توسط کاربر نیز تعریف کرد.

* سربار گذاری عملگر ها فقط برای کلاس ها و ساختارها قابل دسترس است.
* می توان یک عملگر \* برای اینکه با این کلاس یا ساختار استفاده شود، با تعریف کردن متدی با نام operator \* سربار گذاری کرد که رفتارهای آن را پیاده سازی کند ( برای مثال، operator +، operator -، و به همین ترتیب).
  + متد های سربار گذاری شده برای عملگر های یگانی تنها دارای یک پارامتر از نوع کلاس یا ساختار است.
  + متد های سربار گذاری شده برای عملگر های باینری دارای دو پارامتر است، که حداقل یکی از آنها باید از نوع کلاس یا ساختار باشد.

|  |
| --- |
| public static LimitedInt operator -(LimitedInt x) // یگانی  public static LimitedInt operator +(LimitedInt x, double y) // باینری |

برای سربار گذاری عملگر موارد زیر مورد نیاز است:

* در تعریف باید از هر دو اصلاحگر static و public استفاده شود.
* عملگر باید عضوی از کلاس یا ساختاری باشد.

به عنوان مثال، کد زیر سربار گذاری دو عملگر (عملگر جمع و عملگر منفی) را برای کلاس LimitedInt نشان می دهد. می توان گفت که این منفی است و تفریق نیست، زیرا متد تنها یک پارامتر دارد و بنابراین یک عملگر یگانی است در حالی که عملگر تفریق یک عملگر باینری است.

|  |
| --- |
| class LimitedInt Return  { ضروری نوع کلمه کلیدی عملگر عملوند ها  ↓   ↓ ↓ ↓ ↓  public static LimitedInt operator + (LimitedInt x, double y)  {  LimitedInt li = new LimitedInt();  li.TheValue = x.TheValue + (int)y;  return li;  }  public static LimitedInt operator - (LimitedInt x)  {  // در این کلاس عجیب، منفی با مقدار صفر تعیین می شود.  LimitedInt li = new LimitedInt();  li.TheValue = 0;  return li;  }  ...  } |

محدودیت در سربار گذاری عملگر ها

همه عملگر ها می توانند سربار گذاری شوند، و آنها در نوع های سربار گذاری که می توانند انجام شوند محدودیت دارند. مواردی که مهم است درباره محدودیت عملگر ها بدانید در این بخش شرح داده شده اند.

فقط عملگر های زیر را می توان سربار گذاری کرد. مشخص است که عملگر انتساب لیست نشده است.

* قابلیت سربار گذاری عملگر های یگانی : + و - و ! و ~ و ++ و -- و true و false
* قابلیت سربار گذاری عملگر های باینری : + و - و \* و / و % و & و | و ^ و << و >> و == و != و > و < و >= و <=

عملگر های افزایش و کاهش قابل سربار گذاری هستند. اما برخلاف نسخه از پیش تعریف، هیچ تفاوتی بین پس و پیش بودن این عملگر ها وجود ندارد.

نمی توان موارد زیر را با سربار گذاری عملگر ها ایجاد کرد:

* ایجاد یک عملگر جدید
* تغییر ساختار یک عملگر
* تعریف مجدد یک عملگر در نوع های از پیش تعریف شده
* تغییر اولویت یا شرکت پذیری عملگر

|  |
| --- |
| * توجه عملگر های سربار گذاری شده باید معنایی بصری از آن عملگر را داشته باشند. |

مثالی از سربار گذاری عملگر ها

مثال زیر سه عملگر ( منفی، تفریق، جمع) را برای کلاس LimitedInt نشان می دهد.

|  |
| --- |
| class LimitedInt {  const int MaxValue = 100;  const int MinValue = 0;  public static LimitedInt operator -(LimitedInt x)  {  // در این کلاس عجیب، منفی یک با مقدار صفر تعیین می شود.  LimitedInt li = new LimitedInt();  li.TheValue = 0;  return li;  }  public static LimitedInt operator -(LimitedInt x, LimitedInt y)  {  LimitedInt li = new LimitedInt();  li.TheValue = x.TheValue - y.TheValue;  return li;  }  public static LimitedInt operator +(LimitedInt x, double y)  {  LimitedInt li = new LimitedInt();  li.TheValue = x.TheValue + (int)y;  return li;  }  private int \_theValue = 0;  public int TheValue  {  get { return \_theValue; }  set  {  if (value < MinValue)  \_theValue = 0;  else  \_theValue = value > MaxValue  ? MaxValue  : value;  }  }  }    class Program {  static void Main() {  LimitedInt li1 = new LimitedInt();  LimitedInt li2 = new LimitedInt();  LimitedInt li3 = new LimitedInt();  li1.TheValue = 10; li2.TheValue = 26;  Console.WriteLine(" li1: {0}, li2: {1}" ,  li1.TheValue, li2.TheValue);  li3 = -li1;  Console.WriteLine("-{0} = {1}" ,  li1.TheValue, li3.TheValue);  li3 = li2 - li1;  Console.WriteLine(" {0} - {1} = {2}" ,  li2.TheValue, li1.TheValue, li3.TheValue);  li3 = li1 - li2;  Console.WriteLine(" {0} - {1} = {2}" ,  li1.TheValue, li2.TheValue, li3.TheValue);  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| li1: 10, li2: 26  -10 = 0  26 - 10 = 16  10 - 26 = 0 |

عملگر typeof

هر نوع که به عنوان پارامتر به عملگر typeof داده می شود شی ای از نوع System.Type را برمی گرداند. از این شی می توان ویژگی های نوع را فهمید. ( فقط یک شی System.Type برای هر نوع داده شده وجود دارد.) نمی توان عملگر typeof را سربار گذاری کرد. جدول 17-8 ویژگی های این عملگر را لیست کرده است.

عملگر typeof یک عملگر یگانی است.

جدول 17-8. عملگر typrof

|  |  |
| --- | --- |
| عملگر | شرح |
| typeof | از یک نوع داده شده، شی ای از نوع System.Type را برمی گرداند. |

در زیر مثالی از ساختار عملگر typeof وجود دارد. Type کلاسی در فضای نام System است.

|  |
| --- |
| Type t = typeof ( SomeClass ) |

به عنوان مثال، کد زیر از عملگر typeof برای بدست آوردن اطلاعات کلاس SomeClass استفاده کرده است و فیلد ها و متد های عمومی این کلاس را چاپ می کند.

|  |
| --- |
| using System.Reflection; /\* از اطلاعات تعیین شده Reflectionبا استفاده از فضای نام  درباره یک نوع استفاده می شود \*/  class SomeClass  {  public int Field1;  public int Field2;  public void Method1() { }  public int Method2() { return 1; }  }  class Program  {  static void Main()  {  Type t = typeof(SomeClass);  FieldInfo[] fi = t.GetFields();  MethodInfo[] mi = t.GetMethods();  foreach (FieldInfo f in fi)  Console.WriteLine("Field : {0}" , f.Name);  foreach (MethodInfo m in mi)  Console.WriteLine("Method: {0}" , m.Name);  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| Field : Field1  Field : Field2  Method: Method1  Method: Method2  Method: ToString  Method: Equals  Method: GetHashCode  Method: GetType |

عملگر typeof توسط متد GetType نیز فراخوانی می شود، که برای هر شی از هر نوع در دسترس است. به عنوان مثال، کد زیر نام نوع مورد نظر را بدست می آورد.

|  |
| --- |
| class SomeClass  {  }  class Program  {  static void Main()  {  SomeClass s = new SomeClass();  Console.WriteLine("Type s: {0}" , s.GetType().Name);  }  } |

این کد خروجی زیر را تولید می کند:

|  |
| --- |
| Type s: SomeClass |

دیگر عملگر ها

عملگر ها بررسی شده در این فصل عملگر های استاندارد برای نوع های ساخته شده هستند. عملگر های خاص دیگری وجود دارند که بعداً با انواع عملوند های خود در این کتاب بررسی می شوند. به عنوان مثال، نوع های nullable، یک عملگر خاص به نام عملگر تهی به هم آمیخته[[11]](#footnote-11) دارند، که در فصل 25 همراه با نوع های nullable بررسی شده اند.

1. operator [↑](#footnote-ref-1)
2. operand [↑](#footnote-ref-2)
3. expression [↑](#footnote-ref-3)
4. side effect [↑](#footnote-ref-4)
5. escape sequence [↑](#footnote-ref-5)
6. remainder [↑](#footnote-ref-6)
7. shallow comparison [↑](#footnote-ref-7)
8. deep comparison [↑](#footnote-ref-8)
9. Delegates [↑](#footnote-ref-9)
10. enum [↑](#footnote-ref-10)
11. null coalescing operator [↑](#footnote-ref-11)