Operatory

Spis treści

Zapiski uczestnika Bootcampu Zajavka.pl w 12 tygodni by Bartek Borowczyk aka Samuraj Programowania. Dopiski na zielono od Karola Rogowskiego.

Zaczęliśmy naukę operatorów. Mam nadzieję, że te notatki pomogą Wam uzupełnić proces nauki rozpoczęty w lekcjach Karola ☺.

Operatory arytmetyczne

No co tu pisać ☺. Podstawy matematyki. Dodawanie, odejmowanie, mnożenie, dzielenie. Ale nie tylko...

Najpierw zwróćmy uwagę na ciekawe przykłady Karola z filmów.

```
double d = 4 / 3;  ①

double e = (double) 4 / 3; ②
```

① Wyrażenie 4/3 zwraca 1 (bo zwraca int gdy dzielimy dwie wartości int!) i zapisze je w zmiennej d jako typ double (a więc 1.0).

② Rzutujemy pierwszą liczbę double (czyli 4.0) i mamy w związku z tym wynik który jest doublem czyli 1.3333333333333333 i tyle zapisujemy do zmiennej.

Warto jeszcze poznać **modulo**, który oznacza resztę z dzielenia. Jaka jest reszta z dzielenia 8 przez 2 (czyli w programowaniu 8/2 a w matematyce 8:2)? Reszta, oczywiście, wynosi 0. A z 9 przez 2? Tu reszta wynosi jeden.

```
int d = 4 % 3; // wynik to 1
int e = 900 % 400; // wynik to 100
int f = 1001 % 1001; // wynik to 0
```

Promocja numeryczna

Pamiętaj, że przy dodawaniu dwóch liczb wynik dodawania jest podnoszony do wyższego typu. Zerknij na przykłady:

```
byte d = 10;
short e = 200;
short f = d + e; ①
```

① Błąd! Pamiętaj, że w wyniku dodawania byte do short zwracany jest typ int, nawet jeśli taki typ nie jest potrzebny, by przechować powstałą wartość.

Teraz dobrze:

```
byte d = 10;
short e = 200;
int f = d + e; // użyliśmy int
```

Albo teraz:

```
byte d = 10;
short e = 200;
short f = (short) (d + e); // wynikiem dodawania będzie int ale rzutujemy go na short.
```

Podobna sytuacja ma miejsce, gdy dodajemy `int` do `long`. Wtedy zwracana wartość jest longiem.

```
int d = 10;
long e = 200;
int f = d + e; błąd! cannot convert from long to int
```

W takim wypadku mamy dwie możliwości albo zamiast `int f`, tworzymy `long f`:

```
int d = 10;
long e = 200;
long f = d + e;
```

Albo rzutujemy wynik `d + e` na `int` jak poniżej:

```
int d = 10;
long e = 200;
int f = (int) (d + e);
```

Promocja numeryczna dotyczy też typów zmiennoprzecinkowych.

Karol wrzucił ściągawkę, jednak nie musisz się jej uczyć na pamięć, bo z czasem będziesz to czuć.

- Jeśli 2 wartości są innego typu, Java automatycznie wypromuje jedną z nich do wyższego typu,

- Jeśli jedna z nich nie jest zmiennoprzecinkowa, a druga jest, to ta pierwsza zostanie wypromowana do typu zmiennoprzecinkowego,
- Mniejsze typy danych, a mianowicie `byte`, `short` i `char`, są najpierw promowane do wartości int za każdym razem, gdy są używane z binarnym operatorem arytmetycznym Java (np. dodawanie), nawet jeśli żaden z operandów nie jest `int`,
- Po przeprowadzeniu promocji oraz jeśli operandy mają ten sam typ danych, wynikowa wartość będzie miała ten sam typ danych, co jej promowane operandy.

Przy tej okazji warto poznać operatory, które możemy połączyć z operatorem przypisania, a więc:

```
a = a + 1; // możemy też zapisać jako:
a += 1; // co oznacza weź wartość po lewej (a) i dodaj ją do prawej (1) a potem przypisz do a.

a = a * 3; // możemy zapisać jako:
a *= 3;
```

By dodać lub odjąć jedynkę od wartości, możemy też użyć inkrementacji lub dekrementacji.

```
a++; // oznacza dodaj jeden do a, czyli a = a + 1; jest to inkrementacja
a--; // oznacza odejmij 1 od a, czyli a = a - 1; Mamy tutaj dekrementację.
```

Operatory porównania

Zwracają typ `bool`, czyli `true` lub `false`. (Dla jasności, "bool" to taka forma skrócona w mowie, "typ bool", tak się mówi, ale pełna nazwa to `boolean`, przynajmniej w Javie)

```
==  równe   // nie używaj ich do stringów, tam używaj metody equals.
!=  różne od (nierówne)
>   większe
<   mniejsze
>=  większe lub równe
<=  mniejsze lub równe
```

```
System.out.println(2 == 2.2); // false
System.out.println(5 > 4); // true
System.out.println(100 >= 100); // true
System.out.println(1 != 1); // false
```

Operatory logiczne

! - negacja

Najprościej to odwrócenie wartości logicznej na przeciwną (mamy tylko `true` i `false`) i tak: `!true` zwróci `false`, a `!false` zwróci `true`.

```
System.out.println(!(2 > 3)); // 2 > 3 zwróci false, negacja zwróci true, więc wydrukowane będzie true
System.out.println(!true); // false
int a = 10;
long b = 20;
System.out.println(!(a != b)); // !(true) czyli false
```

& - koniunkcja (inaczej iloczyn logiczny)

&& - koniunkcja zoptymalizowana (zwana "na skróty")

Wartości z obu stron muszą być prawdziwe (`true`), by zwrócone zostało `true`. W każdym innym warunku zwrócone zostanie `false `.

```
System.out.println(true & true); // true
System.out.println(true & false); // false
System.out.println(false & true); // false
System.out.println(false & false); // false
```

O co chodzi z tym "na skróty" przy operatorze `&&`. Jest to proces optymalizacji. Przy `&&` oba warunki muszą być prawdziwe. Co oznacza, że jak pierwszy warunek byłby `false`, to sprawdzanie drugiej wartości nie jest konieczne. W zdecydowanej większości przypadków właśnie o to nam chodzi. Proces optymalizacji nie wpływa na zwracany wynik! Po prostu nie wykona kolejnego sprawdzenia, jeśli nie ma to już sensu.

Dlaczego więc zawsze nie używać "na skróty"? Bo czasami chcemy, by drugi warunek (który może być np. wykonaniem metody) był wykonany tak czy siak, bo coś robi. Przekonasz się w praktyce! Na teraz spokojnie, nie przejmuj się, jeśli jest to zbyt teoretyczne, to jedna z tych rzeczy, z którą z czasem się zaprzyjaźnisz.

```
System.out.println(true && true); // true
System.out.println(true && false); // false
System.out.println(false && true); // false - TUTAJ NIE ZOSTANIE SPRAWDZONA DRUGA WARTOŚĆ
System.out.println(false && false); // false - TUTAJ NIE ZOSTANIE SPRAWDZONA DRUGA WARTOŚĆ
```

Przykład z życia ☺.

By jeździć samochodem, trzeba mieć samochód i prawo jazdy (tak, wiem, że niektórzy nie mają a jeżdżą ☺)

Czy może jeździć samochodem?

```
Ma prawo jazdy & ma samochód // sprawdza oba i zwraca true
Ma prawo jazdy && ma samochód // sprawdza oba i zwraca true

Ma prawo jazdy & nie ma samochód // sprawdza oba i zwraca false, bo drugi warunek jest niespełniony
(drugi warunek to false)
Ma prawo jazdy && nie ma samochód // sprawdza oba i zwraca false, bo drugi warunek jest niespełniony
(drugi warunek to false)

Nie ma prawa jazdy & ma samochód // sprawdza oba i zwraca false, bo pierwszy warunek jest niespełniony
(drugi warunek to true)
Nie ma prawa jazdy && ma samochód // sprawdza w tym wypadku pierwszy i widzi false. Ponieważ warunek nie
może być już prawdziwy, nie sprawdza drugiej wartości i zwraca false (czyli optymalizuje).

Nie ma prawa jazdy & nie ma samochód // sprawdza oba i zwraca false
Nie ma prawa jazdy && nie ma samochód // sprawdza w tym wypadku pierwszy i widzi false. Ponieważ warunek
nie może być już prawdziwy, nie sprawdza drugiej wartości i zwraca false (czyli optymalizuje).
```

Pamiętaj że, pisząc "sprawdza", mam na myśli "wykonanie kodu" (to może być jakaś metoda), która zwraca `true` lub `false`.

|- alternatywa (inaczej suma logiczna)

|| - alternatywa zoptymalizowana (zwana "na skróty")

Alternatywa oznacza, że tylko jedna z wartości, obojętnie która, musi zwracać true, by całość była prawdziwa. Oczywiście, warunek jest prawdziwy także, gdy obie wartości są prawdziwe.

```
System.out.println(true | true); // true - sprawdza oba
System.out.println(true | false); // true - sprawdza oba
System.out.println(false | true); // true - sprawdza oba
System.out.println(false | false); // false - sprawdza oba

System.out.println(true || true); // true - sprawdza pierwszy tylko
System.out.println(true || false); // true - sprawdza tylko pierwszy
System.out.println(false || true); // true - sprawdza oba,
// bo pierwszy jest false dlatego by sprawdzić warunek musi sprawdzić drugą wartość
System.out.println(false || false); // false - sprawdza obie
```

^- alternatywa rozłączna (inaczej wykluczająca czy xor)

Taki rzadko stosowany twór (operator). Definiuje się go jako albo jedno, albo drugie, czyli jeśli jedna wartość jest prawdziwa, a druga nie (nieważne, z której strony), to zwracane jest `true`. Gdy obie wartości są nieprawdziwe lub obie są prawdziwe, zwracane jest `false`. Może zastanawiasz się, czemu nie ma `^^` alternatywy rozłącznej "na skróty". To wynika to z tego, że zawsze trzeba sprawdzić oba, nie ma więc tu co optymalizować.

```
System.out.println(true ^ true); // false
System.out.println(true ^ false); // true
System.out.println(false ^ true); // true
System.out.println(false ^ false); // false
```

Operatory bitowe

Tylko na typach numerycznych (tylko całkowitych). Daje on dostęp do bitów, które tworzą daną liczbę.

Nie będą nam potrzebne, o czym Karol mówi po kilkunastu minutach tłumaczenia, czym są i jak działają 😉.

Złożone operatory przypisania

Zacznijmy od tego, co może wydawać się dziwne, a jest bardzo proste 😉.

```
wartość--   postdekrementacja (dekrementacja)
wartość++   postinkrementacja (inkrementacja)
++wartość   preinkrementacja
--wartość   predekrementacja
```

Po pierwsze, inkrementacja i dekrementacja znaczą zwiększ o 1 (inkrementacja) i zmniejsz o 1 (dekrementacja). Takie działania występują powszechnie w programowaniu, dlatego zapis `x++` czy `x--` jest bardzo popularny (i jego odmiana `++x` i `--x`). To jest najważniejsze do zapamiętania. Dodam, że to nie jest wymysł Javy, zapis ten występuje w wielu językach programowania.

Istnieje drobna różnica między `++x` i `x++`, czyli między preinkrementacją a postinkrementacją (i to samo z dekrementacją), którą warto dostrzegać i rozumieć.

Inkrementacja, zwana też postinkrementacją (x++) i preinkrementacja(++x). Zobaczmy:

```
int a = 1;
a++; // zwiększamy o jeden, czyli a = a + 1, a ma teraz 2
 System.out.println(a); // wydrukuje 2

int b = 1;
++b; // zwiększamy o jeden, czyli b = b + 1, b ma teraz 2
System.out.println(b); // wydrukuje 2
```

Jak widać w przykładzie wyżej, efekt jest dokładnie taki sam, gdzie ta różnica? Zobaczmy na drugi przykład.

```
int a = 1;
System.out.println(++a); // wydrukuje 2
int b = 1;
System.out.println(b++); // wydrukuje 1!
System.out.println(b); // wydrukuje 2
```

Różnica między zapisem `x++` a `++x` jest w kolejności działań, które wykona program.

- `x++` - oznacza zwróć wartość x a następnie dodaj do niej 1.
- `++x` - oznacza dodaj do wartości 1 a następnie zwróć tą wartość.

Taki zapis ma znaczenie tylko, gdy wykonujemy jakieś działanie w danej instrukcji. W kolejnej wartość x, niezależnie, czy użyjemy pre czy postinkrementacji, będzie zwiększona o 1.

```
int a = 0;
int b = a + a++;
System.out.println(a); // a ma 1, wydrukowane zostanie 1
System.out.println(b); // b ma 0, wydrukowane zostanie 0
```

Pewnie Cię to zaskoczy, ale zobaczmy:

```
int a = 0;
int b = a + a++;
// krok 1: b = 0 + 0++
// krok 2: b = 0 - najpierw dokonało się dodawanie,
//   czyli z wartości a++ zostało zwrócone 0 i dopiero po zwróceniu wykonał się krok trzeci,
//   czyli a++ czyli 0++:
// krok 3: a + 1; a = 1 - na tym etapie ma już 1
```

Gdy użyjemy preinkrementacji, efekt będzie inny:

```
int a = 0;
int b = a + ++a;
System.out.println(a); // a ma 1, wydrukowane zostanie 1
System.out.println(b); // b ma 1, wydrukowane zostanie 1
```

Przeanalizujmy:

```
int a = 0;
int b = a + ++a;
// krok 1: b = 0 + ++0
// krok 2: b = 0 + 1 - na tym etapie a ma już wartość 1
// krok 3: b = 1 - na tym etapie b ma już wartość 1
```

Pozwól na jeszcze jeden przykład, który już porusza instrukcję warunkową, ale nie przejmuj się, kod będzie zrozumiały, tylko się wczytaj ☺.

```
int x = 999;

if (x++ == 1000)
    // czy ten warunek jest Twoim zdaniem prawdziwy?
    System.out.println(x);
else if (x++ == 999)
    // a może ten?
    System.out.println(x);
else
    // a może żaden warunek nie był prawdziwy?
    System.out.println(x);
```

Przeanalizujmy:

```
int x = 999; // x ma 999 ;)

    // W momencie porównania x wynosi 999!
    // dopiero po porównaniu wartości x z wartością 1000
    // (a nie 999 nie jest równe 1000) przypisane będzie 1 do x.
    if (x++ == 1000)
        System.out.println(x);
        // kod się nie wykona, bo if nie był prawdziwy,
        // ale gdyby się wykonał, to na tym etapie x pokazałby już 1000

    else if (x++ == 999)
        // tutaj w momencie porównania x ma 1000, tak więc znowu warunek nie jest prawdziwy.
        System.out.println(x);
        // x zwróciło do porównania 1000 ale po porównaniu ma już wartość 1001
    else
        System.out.println(x); // tutaj wydrukuje 1001
```

Operator przypisania

Jak wiesz, znak równości to operator przypisania, oznacza on polecenie dla programu: przypisz do zmiennej po lewej stronie wartości znajdujące się po prawej stronie (np. `x = 2`). Ale operator przypisania może wystąpić też w innych konfiguracjach niż tylko znak równości. Najczęściej będzie to plus lub minus (`+=` i `-=`). Wystarczy, że zapamiętasz i zrozumiesz te przykłady poniżej.

```
+=      -=      *=      /=
```

Oba zapisy poniżej są identyczne, jeśli chodzi o wynik działania:

```
int a = 5;
a = a + 2; // a = 5 + 2, czyli a = 7

int a = 5;
a += 2; // weź wartość elementu po lewej stronie i
    // zrób z nią i wartością po prawej stronie to,
    // co określa operator przy znaku równości,
    // a następnie przypisz do zmiennej po lewej stronie
// krok 1:  5 - weź wartość z lewej strony
// krok 2:  2 - weź wartość z prawej strony
// krok 3: zrób to, co mówi znak po lewej przy znaku równości czyli 5 + 2
// krok 4: przypisz nową wartość do zmiennej po lewej stronie czyli a = 7
```

Choć opis może wydawać się skomplikowany i długi, to kiedy programista widzi taki zapis (`a+=2`) nie ma żadnych wątpliwości, co on oznacza. I ty też sobie z tym poradzisz, jeśli w tym momencie jeszcze jest to mylące.

Przy okazji `+=` działa też na Stringi.

```
String a = "Mam ";
a += "ochotę na Javę!";
System.out.println(a); // Mam ochotę na Javę!
```

Kolejność operatorów

Uwaga, nie ucz się na pamięć, szkoda czasu (chyba, że szybko uczysz się na pamięć, to śmiało). Zapamiętaj po prostu najważniejsze reguły.

```
long a = 200 * (2 + 2) / 5;
System.out.println(a); // 160

// 1. 200 * 4 / 5  - najpierw to co w nawiasie
// 2. 800 / 5 - dzielenie i mnożenie mają ten sam priorytet,
    // przy tym samym priorytecie działania wykonują się od lewej do prawej.
```

., **()** **i** **[]** - czyli kropka i działania w nawiasach wykonują się w pierwszej kolejności - mają najwyższy priorytet - to może być więc wykonanie funkcji, zgrupowane w nawiasach wyrażenia oraz pobranie wartości z obiektu.

A następnie: **(pre)inkrementacja (++) i (pre)dekrementacja (--)**

I dalej (w tej samej linii prezentowane operacje o tym samym priorytecie a czym niżej, tym niższy priorytet i późniejsze wykonanie):

Tabela 1. Operatory i kolejność wywołania:

Operator
mnożenie (*) i dzielenie (/) oraz modulo - czyli reszta z dzielenia (%)
dodawanie (+) i odejmowanie (-)
> < >= ⇐ (porównanie)
== i != (równe i różne)
& (koniunkcja - iloczyn logiczny czyli 'i' po angielsku and)
^ (alternatywa rozłączna)
| (alternatywa - suma logiczna czyli "lub" po angielsku or)
&& (koniunkcja zoptymalizowana)
|| (alternatywa zoptymalizowana)
=, += -= (itp.) przypisanie

Najlepiej to przećwiczyć na przykładach:

Przykład 1

```
5 < 2 || 2 < 1 * 5  // true
```

Jak to wygląda w praktyce:

Krok 1: mnożenie, czyli:

```
5 < 2 || 2 < 5
```

Krok 2: porównanie w kolejności od lewej do prawej:

```
false || 2 < 5
false || true
```

Krok 3: suma logiczna (przypominam, jedna wartość musi być prawidłowa, by zwrócić `true`). Pierwsza wartość nie jest u nas prawdziwa, więc sprawdzana jest też druga, a druga jest `true`, więc całość jest również `true`.:

```
true
```

Przykład 2

I jeszcze jeden przykład:

```
int a = 1;

a != 0 & 2 < "a".length(); // false
```

Krok 1: wykonanie metody `length()`:

```
a != 0 & 2 < 1;
```

Krok 2: porównanie:

```
a != 0 & false
```

Krok 3: operator różne od:

```
true & false
```

Krok 4: operator koniunkcji - wartość z lewej i prawej muszą być wartościami `true`, by operacja zwróciła `true`, inaczej zwróci `false`.

```
false
```

Ta kolejność wchodzi do głowy szybko, kiedy piszesz kod, dlatego niekoniecznie mas sens się jej uczyć w tym momencie. Zapamiętaj, że jest podobnie jak w matematyce i ma to swoją logikę ☺. Potwierdzam, jest dokładnie tak jak Bartek napisał. ☺ Zerknij na to porównanie: link.

Kolejność operatorów bitowych pominę, bo nie mają one wielkiego znaczenia (i operatory i ich kolejność) - przynajmniej dla 99% programistów. A jeśli zdarzy się, że w przyszłych projektach w pracy byś ich potrzebował, to na pewno wtedy sięgniesz po informacje z internetu. Na teraz nie musisz o nich myśleć ☺.