Zmienne

Spis treści

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

Co to jest zmienna?

Zmienna to podstawowa struktura programistyczna we wszystkich popularnych językach programowania.

Zmienna składa się zawsze z nazwy i typu oraz najczęściej przechowuje jakąś wartości lub obiekt. W istocie przechowuje nie samą wartość, a odnośnik do pamięci, gdzie dana wartość się znajduje (ale potocznie możemy powiedzieć, że przechowuje daną wartość).

Dzięki zmiennym możemy przechowywać dane i operować na danych w programie.

Deklaracja zmiennych

Zmienną możemy zadeklarować, wskazując dwa elementy:

- typ przechowywanych danych (np. któryś z typów prostych, które poznaliśmy)
- nazwę, która nie może być nazwą zastrzeżoną (np. niepoprawne jest użycie składni `int int` czy `int class`) i musi zaczynać się od litery (czyli `1value` nie jest prawidłowe, a `value1` już tak)

Przykładowe poprawne deklaracje

```
int customerNumber;
boolean hasPermission;
```

Konwencja mówi też, że używamy notacji wielbłądziej, ale możesz używać też zamiast niej np. podkreśleń czyli nazwa `has_permission` jest ok.

Jeśli deklarujemy zmienne tego samego typu, to możemy to zrobić w jednym wierszu:

```
int number1, number2, number3;
double a,b,c,d;
```

Na końcu deklaracji zawsze umieszczamy średnik!

Inicjalizacja zmiennej

Polega na przypisaniu wartości do zmiennej.

W poprzednim przykładzie stworzyliśmy (zadeklarowaliśmy) 4 zmienne:

```
double a,b,c,d;
```

Inicjalizujemy je, przypisując im wartość, do czego służy operator przypisania reprezentowany przez znak równości.

```
double a,b,c,d;
a = 3;
b = 4.32423;
c = 1999999.1;
d = 0;
```

Bardzo często razem z deklaracją robimy też jednocześnie inicjalizację, czyli nadajemy wartość danej zmiennej.

```
int age = 56;
boolean isProgrammer = true;
```

W tym momencie to wszystko, co musimy wiedzieć o zmiennych.

Pamięć a zapisywane wartości

Wartości używane w programie są zapisywane w pamięci. Program korzysta z pamięci RAM (jest to pamięć tymczasowa).

Podstawowa jednostka w pamięci jest reprezentowana przez **bajt, a więc 8 bitów**.

Polecam ten film o działaniu procesora, pamięci, kodzie maszynowym oraz przekształceniu liczb binarnych na dziesiętne i w drugą stronę. Tym bardziej polecam, że jestem jego autorem ☺. Ale warto go obejrzeć, bo tłumaczy sporo, link.

Typy prymitywne (inaczej proste, podstawowe)

Typy całkowite

Zacznijmy od typów całkowitych (liczby całkowite, typ całkowitoliczbowy).

byte

byte - składa się z 1 bajta (ang. `byte`), czyli podstawowej, minimalnej wielkości pamięci. Bajt, jak pokazywał Karol, tworzy 8 bitów, które mogą przyjąć wartość 0 lub 1 (czyli system dwójkowy). Za pomocą byte możemy zapisać wartości od `-128` do `127`. Np. wartość `3` zostanie zapisana jako `00000011` a wartość `100` jako `01100100`.

short

short - tutaj zawsze wykorzystane będą dwa bajty. Oczywiście, w shorcie, jak i innych typach numerycznych, można przechować także wartość, np. `0`, przy czym wtedy zajmiemy i tak dwa bajty (`00000000 00000000`). Short może pomieścić wartości od `-32 768` do `32 767`. Ps. nie ucz się tego na pamięć!

int

int - tworzą go `4` bajty a możliwy zakres przechowywanych liczb to od `-2147483648` do `2147483647`.

long

long - od `-9223372036854775808` do `9223372036854775807`. W 99.999 proc przypadków nie potrzebujesz większej liczby niż tryliard ☺, `long` korzysta z 8 bajtów. Long wystarczy nawet do zapisania prawdopodobnej liczby gwiazd we wszechświecie widzialnym (tak więc, gdyby świat był symulacją, to liczbę gwiazd zmieścilibyśmy w 4 bajtach ☺). Typ long wymaga ponadto użycia litery **l** lub **L** (małe lub wielkie **L**) - choć możemy użyć obu, to Karol rekomenduje wielkie, bo, jak widać, małe **l** może się mylić z **1**.

Na pierwszy rzut oka wydaje się, że do większości zastosowań nadaje się `int`, ale biorąc pod uwagę, że 4 bajty to naprawdę mało, to czy nie korzystać z `long` na wszelki wypadek (`byte`, `short` niemal wcale, a `int` tylko, jeśli na 100% wiemy, że w dalekiej przyszłości potrzeby się nie zmienią)? - To pytanie kieruje do Karola i mam nadzieję, że tu albo w przyszłości usłyszymy odpowiedź ☺. Generalnie w praktyce należy się

> zawsze zastanowić nad możliwym zakresem wartości. Jeżeli już jesteś pewien, że Twoja wartość się nie zmieści, to używaj `long`. Przykładowo, na co dzień do przechowywania *id klienta*, na którym operujemy, stosuje się `long`. W przyszłych filmach będę też mówił o czymś takim, jak `BigInt` i `BigDecimal`, gdyby podane zakresy były za małe, albo potrzebowalibyśmy dużej dokładności obliczeniowej.

Liczby możemy zapisywać także w innych systemach niż najbliższy ludziom dziesiętny, choć, oczywiście, najczęściej używa się systemu dziesiętnego (ang. decimal). Tak więc w Javie prawidłowym będzie zapisanie liczby 10 także w innych systemach.

Liczba 10 zapisana w Javie za pomocą innych systemów liczbowych:

```
szesnastkowym: 0xA (przedrostek 0x // 0 to zero)
ósemkowym: 012  (przedrostek 0)
dwójkowym (binarnym): 0b1010 (przedrostek 0b)
```

Przykład deklarowania i inicjalizowania zmiennych typu liczb całkowitych

```
byte number1 = -10;
short number2 = 12280;
int number3 = -1240239; // możemy też zapisać -1_240_239
long number4 = 23948239234L; // możemy też zapisać 23_948_239_234L
```

Typy zmiennoprzecinkowe

Typ zmiennoprzecinkowy pozwala przechowywać wartości zmiennoprzecinkowe (czyli ułamki). W Javie występują dwa takie typy:

float

- **float** - jego wielkość to 4 bajty,

double

- **double** - bardziej precyzyjny (double - podwójnie precyzyjny) - przechowuje wartość ułamkową za pomocą 8 bajtów (czyli 64 bitów). Powinien być traktowany jako domyślny typ do pracy z wartościami ułamkowymi.

Jeśli korzystamy z `float`, to po liczbie powinniśmy użyć znaku **f** lub **F**. Przyrostka nie potrzebujemy przy double, choć możemy użyć go także tu i będą to znaki **d** lub **D**.

Przykład deklarowania i inicjalizowania zmiennych:

```
float number1 = 2.390723F;
double number2 = 2.43;
double number3 = 1 // warto wiedzieć, że i tak zapisywane/odczytywane jest jako 1.0
```

Pozostałe

boolean

boolean - Ja wymawiam "*bulin*", Karol widzę podobnie ☺. Zdarza się wymowa "*bulen*". Przyjmuje on jedną z dwóch wartości: `prawda` lub `fałsz`, a właściwie `true` lub `false`, pisane małą literą! W niektórych innych językach można też używać `0` lub `1`, tutaj nie! ☺

```
boolean value1 = true;
boolean value2 = false;
```

char

char - wymawiamy "czar" lub "kar" (wtedy od słowa "character", czyli znak) ☺. Bo typ char to właśnie pojedynczy znak. Specyfiką typu char jest np. to, że nie możemy tu zastosować cudzysłowu (" "), a musimy posłużyć się apostrofem (' ').

```
char elementB = 'B';
System.out.println(elementB);
```

Wydrukuje nam **B**, oczywiście.

Ale możemy użyć też liczby, która reprezentuje dany kod w **ASCII** lub **Unicode**:

```
char elementB = 66;
System.out.println(elementB);
```

Liczba 66 w tablicy znaków `ASCII` reprezentuje właśnie znak **B**. Warto wiedzieć, że w tablicy *ASCII/Unicode* litery wielkie i małe mają przypisane inne liczby, np. małe 'b' jest reprezentowane przez liczbę 98.

Pokrótce o samym ASCII i Unicodzie. Procesor pracuje na liczbach, ale my chcemy używać też liter i innych znaków. Jak to zrobić? Właśnie za pomocą kodowania, czyli przypisywania jakiemuś znakowi liczby, która go reprezentuje. Tak powstało ASCII a później Unicode. Unicode jest rozszerzoną wersją ASCII, ASCII bowiem nie zawiera liter i znaków z innych alfabetów. Jeśli chcemy uzyskać taki znak, np. małe polskie 'ł', to musimy posłużyć się liczbą z tablicy Unicode (czyli większym numerem niż daje ASCII). Java i typ char, oczywiście, sobie z tym poradzą.

```
char character = 322;
System.out.println(character);
```

W powyższym przypadku zwrócone zostanie 'ł'.

Nie potrzebujemy w tym momencie wiedzieć nic więcej o ASCII i Unicode ani tym bardziej o sposobie kodowania UTF-16 ☺. Zresztą na co dzień też nie musisz się kompletnie przejmować wartościami znaków w Unicodzie czy ASCII (przy okazji, ASCII wymawiamy jak "aski"). A i korzystanie z typu char też jest bardzo, bardzo rzadkie, ponieważ korzystamy z czegoś takiego, jak łańcuch (String), o czym na pewno dowiemy się w przyszłości ☺.

Co jeszcze warto wiedzieć

Java posiada ścisłą kontrolę typów, tzn. wartość przechowywana w zmiennej musi mieć określony, zadeklarowany typ.

W nazwach można używać liter z tablicy Unicode, a więc poprawne jest użycie w nazwie (np. zmiennej) polskich liter. Natomiast konwencja jest taka, że tego nie robimy. Nazwy piszemy po angielsku.

Liczby zmiennoprzecinkowe (`float`, `double`) nie są dobrym rozwiązaniem do zapisywania wartości pieniężnych i operacji finansowych. Sposób zapisu liczby w pamięci (w systemie binarnym) powoduje, że nie ma tu wystarczającej precyzji w systemie dziesiętnym.

```
System.out.println(1.1 - 0.2);
```

Wydrukuje: 0.9000000000000001

```
System.out.println(1.2 - 0.1);
```

Wydrukuje: 1.0999999999999999

Karol wspominał, że używa się tutaj klasy `BigDecimal`, która z tym "błędem" sobie radzi - i o tym pewnie dowiemy się w przyszłości.

Final

final - stałe. W Javie istnieje co prawda takie słowo jako `const` (`const` od constant - stała), ale nie jest ono do niczego używane. Wspominam o tym (i wspominał Karol), ponieważ zdarza się, że const w innych językach służy do konstruowania stałej (w JavaScript na przykład).

W Javie także możemy tworzyć stałe, ale służy do tego słowo kluczowe `final`.

Słowo kluczowe `final` pozwala tworzyć stałą. Co to oznacza? `Final` tworzy trwałe łącze (trwałą referencję) z miejscem w pamięci. I to połączenie nie może być już zmieniane w innym miejscu programu. W praktyce przypisujemy do zmiennej typ prosty (jak przypiszemy `20`, to już zawsze będzie `20`) lub obiekt i taka zmienna zawsze już będzie połączona z tym obiektem, choć sam obiekt może ulec modyfikacji a `final` nie czyni go "zamrożonym" (niemutowalnym, niezmienialnym), co też podkreśla Karol.

```
final int value = 10;
value = 10 + 1; // Błąd
// The final local variable value cannot be assigned.
```

Często dla stałych używamy nazw zbudowanych z wielkich liter, np.:

```
final double MAX_WIDTH = 500.5;
```

Czy `final` to więc ciągle zmienna? Tak, to zmienna, która jest stała ☺. I jakby to dziwnie nie brzmiało, to tak należy to rozumieć. Pamiętając, że chodzi tu o trwałe połączenie z określonym miejscem w pamięci.

Rzutowanie

Rzutowanie to jawna konwersja jednego typu na inny. Bez problemu możemy rzutować typ, który ma mniej bajtów, na typ, który ma więcej bajtów np. `float` na `double` czy `int` na `long`.

Oprócz rzutowania, rozumianego jako jawna konwersja (za pomocą jasnej deklaracji), występuje też konwersja automatyczna.

Przyjrzyjmy się takim instrukcjom:

```
int value1 = 2; // wartość typu int
double value2 = value1; ①
```

① Wartość konwertowana automatycznie na double i przypisana do nowej zmiennej

Wartość przypisana do `value2`, a znajdująca się w `value1`, zostanie zamieniona na `double` (mimo, że przekazujemy `int`). Gdybyśmy chcieli ją odczytać, to byłoby to `2.0`.

W drugą stronę taka automatyczna konwersja nie zadziała.

```
double value1 = 2;
int value2 = value1; ①
```

① błąd: cannot convert from double to int

Typ `double` ma większą liczbę bajtów w pamięci niż `int` (jeśli nie pamiętasz ile, to znajdziesz to we wcześniejszych notatkach ☺). Dlatego kompilator odmawia automatycznej konwersji. Podobnie będzie, gdy spróbujemy konwertować `long` na `int`.

```
long value1 = 2L;
int value2 = value1;
// błąd: cannot convert from long to int
```

Takiej konwersji możemy jednak dokonać jawnie (co nazywamy **rzutowaniem**, z ang. **casting**). Oczywiście, istnieje tutaj ryzyko utraty precyzji (a więc prawidłowych informacji).

Zobaczymy przykłady.

```
long value1 = 29843L;
int value2 = (int) value1;
```

W ten sposób uda nam się zapisać wartość ze zmiennej `value1` (typu `long`) w zmiennej `value2` (typu `int`) i to, w tym wypadku, bez straty, ponieważ wartość przechowywana w `long` da się zamieścić także w `int`. Ale zobaczmy ten przykład:

```
long value1 = 9843495834905833L;
int value2 = (int) value1;
```

Otrzymamy taką wartość w `value2: 2023124201` - czyli na pewno nie taką, jakiej oczekujemy ☺.

Możemy, oczywiście, rzutować także typ zmiennoprzecinkowy na typ całkowitoliczbowy.

```
double value1 = 33.6421;
int value2 = (int) value1;
```

W tym wypadku otrzymamy wartość całkowitą 33, po prostu ułamek zostanie odrzucony. Zwróć uwagę, że nie dokonało się zaokrąglenie, Java zwyczajnie odrzuca to, co było po przecinku.

String

String to typ, który przechowuje ciąg znaków (można go traktować jako zbiór znaków Unicode). Nie jest to jednak typ prosty.

String jest klasą, która, jak wiele innych klas i metod, jest dostarczana przez samą Javę. Mówimy tu więc o rozwiązaniu, które jest dostarczane przez standardową bibliotekę Javy (zwaną też API Javy) w ramach JDK. **API Javy** dostarcza wiele klas i metod i jest dostępne dla programu napisanego w Javie (np. metoda println jest częścią tego API).

```
String bootcampTitle = "Zajavka";
```

`String` zawiera znaki i jest **obejmowany cudzysłowem** (ale nie apostrofem! Apostrof jest przy typie `char`).

Zwróć uwagę na nazwę typu. Jest to **String**. I, jak już wiesz, nie należy on do typów prostych, co sugeruje już sama nazwa pisana dużą literą. To chyba pierwszy przykład w tym Bootcampie, gdzie tworzymy wartość, która jest obiektem, a nie typem prostym.

Czy dwa `Stringi` o tej samej zawartości są takie same? I to jest bardzo dobre pytanie, szczególnie jeśli uczymy się Javy ☺. Takie pytania też padają na rekrutacji, dlatego Karol wprowadził takie pojęcia jak *String pool* i metoda `intern()`, ale o tym na koniec. Wróćmy do pytania, czy Stringi o tej samej zawartości są takie same?

Zacznijmy od porównania za pomocą dwóch znaków równości, czyli "==" (co nie jest dobrym rozwiązaniem, jeśli chodzi o `Stringi`).

Tak więc, jeśli dwa Stringi powstały za pomocą **literału**[1], to ich porównanie za pomocą operatora `==` zwróci nam `true`.

[1] - literał to zrozumiała dla kompilatora konstrukcja znaków, w tym wypadku (Stringów), zaczynająca się i kończąca cudzysłowem, która wprost definiuje jakąś wartość. Inne literały to literał liczbowy, np. `42` czy `true` (literał tworzący typ `boolean`).

Przykład 1

```
String txt1 = "tekst";
String txt2 = "tekst";
System.out.println(txt1 == txt2); ①
```

① wydrukowane zostanie: **true**

Możemy też tworzyć Stringa jak inne obiekty na podstawie odwołania do klasy i **słowa kluczowego new**. Ale tu ciekawostka. Porównanie tych dwóch wartości (obiektów) da już `false`. Dlaczego? Bo wartości tych dwóch obiektów, mimo, że są takie same, trzymane są w różnych miejscach pamięci. W przypadku literałów Java uznaje, że jeśli tworzysz `String` o takiej samej wartości (zawierający te same znaki), który już był tworzony, to optymalizuje to i przypisuje do tego samego miejsca w pamięci. W przykładzie powyżej powstała więc jedna wartość w pamięci, do której prowadzi zarówno pierwsza, jak i druga zmienna. W drugim przykładzie, tym poniżej, mamy też dwa obiekty, ale już dwie różne pozycje w pamięci.

Przykład 2

```
String txt1 = new String("tekst");
String txt2 = new String("tekst");
System.out.println(txt1 == txt2); ①
```

① wydrukowane zostanie: **false**

Pamiętaj, że za pomocą dwóch znaków równości sprawdzamy nie to, czy wartości Stringów są takie same, ale czy prowadzą do tego samego miejsca w pamięci (czyli czy zmienne mają referencje do tego samego miejsca pamięci!) W praktyce więc, mimo że mogą mieć taką samą zawartość, to mogą prowadzić do różnych miejsc pamięci, w zależności, jak zostały utworzone. Można pomyśleć, że to głupie! No cóż ☺.

No dobrze, a co się dzieje tutaj (w przykładzie 3)? Stworzyliśmy nowy obiekt `String`. Następnie przypisujemy zmienną `txt1` (zawierającą referencje do obiektu) do nowej zmiennej typu `String`. I mamy zaskoczenie (albo i nie ☺), są takie same, tzn. wskazują na ten sam obiekt. Jest tylko jeden obiekt i zarówno `txt1` i `txt2` wskazują na niego.

Przykład 3

```
String txt1 = new String("tekst");
String txt2 = txt1;
System.out.println(txt1 == txt2); ①
```

① wydrukowane zostanie: **true**

Częściowo stanie się to zrozumiałe, jeśli potraktujesz zmienną nie jako pojemnik na dane, tylko link do miejsca w pamięci, tzw. **referencję**. Kiedy tworzysz dwa obiekty (dwie instancje, dwa egzemplarze), nawet identyczne, to tworzysz dwa różne obiekty. Dwa różne obiekty nie są tymi samymi obiektami, a to właśnie jest sprawdzane poprzez dwa znaki równości. Stąd `false` w przykładzie drugim. Natomiast w przykładzie pierwszym najpierw stworzyliśmy obiekt, ale kiedy już jeden obiekt w pamięci istnieje, a tworzymy taką samą wartość za pomocą literału, to wskazujemy na tę wartość, która już istnieje. Dodam mały komentarz, że cały czas rozmawiamy o Stringach, żeby ktoś nie pomyślał, że tak jest w przypadku każdego innego rodzaju danych.

```
String txt1 = "tekst"; ①
String txt2 = "tekst"; ②
```

① Stwórz obiekt String z wartości "tekst".

② Wiesz co, mam już taki obiekt z taką wartością w pamięci więc skieruje Cię do tej wartości, zamiast tworzyć nową.

W przykładzie trzecim widzimy jeszcze jedną ważną rzecz. Otóż w pierwszej instrukcji tworzymy obiekt i referencję do niego przypisujemy do zmiennej `txt1`. Natomiast w drugiej instrukcji tworzymy zmienną typu `String` i nie kopiujemy do niej obiektu (nie tworzymy nowej instancji!), a jedynie zapisujemy w zmiennej referencję do miejsca w pamięci, gdzie jest ten obiekt. Obie zmienne są więc referencjami do tego samego obiektu!

Zapamiętaj więc, że sposób z dwoma znakami równości nie jest dobrym sposobem na porównanie Stringów, bo w zależności od tego, jak powstały (a nie wymieniliśmy wszystkich możliwości, np. różnych metod, które przykładowo tworzą z jednego Stringa inny String), rezultat może być inny. Do porównania Stringów najlepszym rozwiązaniem jest **metoda** `equals()`.

```
String txt1 = new String("tekst");
String txt2 = new String("tekst");
System.out.println(txt1 == txt2);  // false
System.out.println(txt1.equals(txt2)); // true
```

Zapamiętaj więc konstrukcję:

```
String1.equals(String2);
```

np.

```
"A".equals('a'); // zwróciłoby false

String numberTXT = "120";
"120".equals(numberTXT); // zwróciłoby true

"A".equals("A"); // zwróciłoby true
```

Metoda `equals()` porównuje nie miejsce w pamięci, a rzeczywistą zawartość obiektu. W przypadku Stringa liczy się więc tutaj, co zawiera, a nie czy jego wartość jest trzymana w tej samej, czy innej zmiennej. Przy czym pamiętajmy, że metoda `equals()` sprawdza, czy obiekty (stringi) są takie same, a nie te same!

Warto wiedzieć, że `String` może przyjąć też takie wartości jak:

- **pusty String**, czyli jedynie cudzysłów "" niezawierający nic w środku.
- wartość **null**, co oznacza, że nie jest do niego przypisane żadne miejsce w pamięci.

Co jeszcze warto wiedzieć o Stringach:

Są niezmienialne (niemutowalne) - Jeśli przekształcamy `String`, to tworzymy tak naprawdę nowy.

String pool - To miejsce w pamięci, do którego trafiają niektóre Stringi. Jest to spowodowane optymalizacją Javy. Przy tworzeniu nowego Stringa za pomocą literału Java sprawdza, czy dany element jest już w *String pool*, czyli czy ma dany String (zawierający dane znaki) w pamięci określanej jako pool (stringi stworzone za pomocą new nie są trzymane w pool). Jeśli tak, to przypisuje zmienną (referencja) do tej wartości.

```
String txt1 = "tekst";
String txt2 = "tekst";  // mechanizm String pool
String txt3 = "tekst";  // mechanizm String pool
String txt4 = new String("tekst"); // nowy obszar pamięci - domyślnie nie korzysta ze String pool
```

Zapamiętaj: Java przy tworzeniu Stringów z literałów (przypominam, że to też obiekty), sprawdza, czy w pamięci nie przechowuje już danego tekstu, a jeśli tak, to powoduje współdzielenie tego tekstu z innym obiektem (optymalizuje w ten sposób pamięć). Jeśli natomiast dodalibyśmy `String` za pomocą konstrukcji new, zawsze tworzone jest nowe miejsce w pamięci, nawet z taką samą wartością.

String intern - metoda intern służy do wymuszania zapisu wartości tworzonego Stringa w String pool - chodzi, oczywiście, o Stringi stworzone z operatorem new.

```
String txt1 = new String("string");
String txt2 = "string";
String txt3 = txt1.intern();

// false, bo zapisane w innym miejscu pamięci
System.out.println(txt1 == txt2);
// true, bo wymusiliśmy zapisanie w pool,
// więc prowadzi do tego samego miejsca w pamięci, w którym znajduje się już txt2
System.out.println(txt2 == txt3);
```

Tutaj może to napiszę, bo Bartek zadzwonił do mnie przy pisaniu tych notatek, żeby mi powiedzieć, że chyba za mocno dorzuciłem do pieca tym String poolem i metodą `intern()` ☺. Chciałbym tutaj napisać, czemu ten temat wjechał. Na rozmowach rekrutacyjnych potrafi paść takie pytanie: wytłumacz, czemu w kodzie poniżej obserwujemy takie zachowanie, jak widać i powiedz co zrobić, żeby zamiast false było drukowane true:

```
String txt1 = "tekst";
String txt2 = new String("tekst");
System.out.println(txt1 == txt2);  // false
System.out.println(txt1.equals(txt2)); // true
```

Bez posiadania wiedzy nt. String pooli i metody intern(), nie ma opcji, że to wytłumaczysz ☺.

Konkatenacja

Łączenie Stringów czy innych typów ze Stringami - to oznacza właśnie pojęcie konkatenacji. Dokonujemy jej za pomocą operatora +. String jest zwracany, jeżeli choć jedna wartość łączona jest Stringiem.

Przykład

```
System.out.println("A" + "B"); // Wynik AB
System.out.println("A" + 2); // Wynik A2
System.out.println(2 + 3 + "A" + 2); // Wynik 5A2
```

Widać, że mamy tutaj kolejność od lewej do prawej (można ją zaburzyć, dodając nawiasy, to co w nawiasach ma pierwszeństwo). Zaskakujący może być przykład trzeci. Widzimy, że łączymy tutaj 2 i 3, czyli w istocie nie łączymy a za pomocą operatora "+" dodajemy dwie liczby, stąd 5 a nie "23". Dopiero w kolejnym działaniu mamy połączenie z 'A'. Czyli jak pojawi się już String (idąc od lewej), to reszta już jest dodawana jako String.

Do konkatenacji możemy też użyć metody `concat` (dostępnej dla Stringów, bo dostarczanej z klasą String).

```
String txt1 = "Mam ";
String txt2 = "chęć ";
String txt3 = "Na Javę!";

String newtxt = txt1.concat(txt2).concat(txt3);
System.out.println(newtxt);  // Mam chęć na javę
```

Metody z klasy String

String jest więc klasą. Co nam to daje? Możliwość wykonania na nim różnych metod, czyli działań. Razem z klasą String dostarczone jest bowiem kilkadziesiąt metod, które można wykonać na tekście (czyli na obiekcie typu String).

.length()

Metoda length zwraca długość Stringa, czyli liczbę znaków w Stringu, oczywiście, uwzględnia białe znaki. Zwracana wartość ma typ `int`.

```
String name = "Władysław";
System.out.println(name.length()); // wydrukuje 9

String name = "";
System.out.println(name.length()); // wydrukuje 0
```

.charAt()

Metoda charAt zwraca element Stringa ze wskazanej pozycji. Zwracany jest typ char, więc jeśli chcielibyśmy go przypisać do zmiennej, to musi być to zmienna `char`.

```
String name = "Dzień dobry";
System.out.println(name.charAt(4)); // wydrukuje ń
char a = name.charAt(4); // przypisze 'ń'
```

.repeat()

Metoda repeat zwraca wartość String, który jest nowym Stringiem składającym się z wielokrotności wartości Stringa, na którym jest wykonywana metoda.

```
String name = "Dzień dobry";
System.out.println(name.repeat(3)); // Wydrukuje "Dzień dobryDzień dobryDzień dobry"
String a = name.repeat(2); // Przypisze do zmiennej Stringa "Dzień dobryDzień dobry"
```

.substring()

Metoda wykonywana na Stringu. Zwraca nowy String wg podanych parametrów. Możemy podać jeden lub dwa parametry. Jeśli jeden, to od którego indeksu i domyślnie do końca Stringa. Jeśli dwa, to pierwszy parametr mówi, od którego indeksu a drugi, do którego (ale bez tego). Pamiętaj, że indeks liczymy od 0 a nie 1.

```
String name = "Jan Maria";
System.out.println(name.substring(3)); // wydrukuje " Maria"
String a = name.substring(1, 7); // zwróci "an Mar"
System.out.println("Zdanie do przerobienia".substring(3, 5)); // wydrukuje "ni"
System.out.println(a); // wydrukuje "an Mar"
```

.concat()

Metoda dokonująca konkatenacji (połączenia) Stringów. Zwraca nowy String będący połączeniem dwóch Stringów.

```
String name = "Jan";
String name2 = "Roman";
System.out.println(name.concat(name2)); // wydrukuje "JanRoman"
String a = name.concat(name2); // zwróci "JanRoman" i zapisze w zmiennej a
System.out.println(name); // wydrukuje "Jan"
System.out.println(name2); // wydrukuje "Roman"
System.out.println(a); // wydrukuje "JanRoman"
```

.contains()

Zwraca `true` lub `false` (a więc typ `boolean`!), sprawdzając, czy String, na którym jest wykonana metoda, zawiera przekazany do metody `String`.

```
String name = "Dzień dobry";
System.out.println(name.contains("obry")); // zwróci true
boolean a = name.contains("zień"); // zwróci false
System.out.println(a); // wydrukuje false
```

.replace()

Zamienia fragment (pierwszy pasujący) w Stringu, na którym jest wykonywana (jeśli, oczywiście, takie

fragment znajdzie):

```
String name = "Hej Javowcy!";
// wyświetli Stringa pierwotnego, ponieważ nie ma w Stringu name tekstu "hej"
// (wielkość liter ma znaczenie). Metoda replace zwraca za każdym razem nowego Stringa,
// nawet jeśli jest taki sam jak ten na którym pracuje.
System.out.println(name.replace("hej", "cześć"));
// tutaj zadziała i do zmiennej a zostanie przypisany nowy String "Cześć Javowcy!"
String a = name.replace("Hej", "Cześć");
System.out.println(name); // wyświetli Hej Javowcy!
System.out.println(a); // wyświetli Cześć Javowcy!
```

.toLowerCase() i toUpperCase()

Metoda wykonana na Stringu zwraca nowy String pisany małymi literami (`toLowerCase`) lub wielkimi (`toUpperCase`). Pamiętajmy, że String pierwotny nie jest zmieniany (bo jest niemutowalny) i nie jest przypisywany nowy. Te dane, na których pracujemy metodami, nie są zmieniane.

```
String name = "ESTONIA";
System.out.println(name.toLowerCase()); // wydrukuje estonia
String a = "Tomasz Nowak".toUpperCase() // do zmiennej 'a' przypisze TOMASZ NOWAK
System.out.println(name); // wydrukuje ESTONIA
System.out.println(a); // wydrukuje TOMASZ NOWAK
```

.trim()

Zwraca nowy `String`, usuwając z niego spacje z początku i końca.

```
String name = "   aa bb cc  ";
System.out.println(name.trim()); // wydrukuje "aa bb cc"
String a = "Tomasz Nowak ".trim(); // zwróci "Tomasz Nowak"
System.out.println(name); // wydrukuje "   aa bb cc  "
System.out.println(a); // wydrukuje "Tomasz Nowak"
```

var

Gdy nie chcemy wskazywać typu podczas deklarowania zmiennej, to od Java w wersji 10 nie musimy. W takim wypadku Java automatycznie stworzy typ (bo typ musi być zawsze! Przy czym typ nadawany jest dynamicznie przy `var`).

Pamiętajmy też, że `var` można stosować tylko w **zmiennych lokalnych**, a więc zmiennych tworzonych w metodach, ale już nie w polach obiektów (ale czym są te pola, to już w programowaniu obiektowym).

```
public class Zajavka {
    var age = 20; // błąd: 'var' is not allowed here
    public static void main() {
        // metoda - tutaj powstają zmienne lokalne
        var age2 = 20; // tutaj jest ok
        // oczywiście w if, for też mogą być zmienne z var bo nadal to zmienne lokalne, bo są w metodzie.
    }
}
```

Przykład działania (pamiętaj, że chodzi o zmienne lokalne)

```
var a = 1; // stworzy int
var b = "2"; // stworzy Stringa
var c = 58345908350834L; // stworzy longa
```

Są zwolennicy i są przeciwnicy tego rozwiązania ☺.

Zasięg i cykl życia zmiennych

Wprowadzamy też pojęcie **scope** (zasięg, zakres) zmiennej.

Java pozwala nam deklarować zmienne w dowolnym **bloku kodu**. Blok kodu to przestrzeń ograniczona nawiasami klamrowymi {}. Jeden blok kodu może być, oczywiście, zagnieżdżony w innym, co dobrze widzimy na tym przykładzie.

```
public class Main {
    // 1 blok kodu
    public static void main(String[] args) {
        // 2 blok kodu
        {
            // 3 blok kodu
        }
    }
}
```

W przyszłości poznasz wiele takich struktur: klasy, metody, pętle, instrukcje warunkowe - to najpopularniejsze z nich.

Blok kodu tworzy zasięg naszej zmiennej. Zmienna stworzona w danym bloku jest widoczna tylko w danym bloku i w blokach potomnych (zagnieżdżonych), ale nie tych nadrzędnych. Druga sprawa to czas życia takiej zmiennej. Otóż zmienna "żyje" w pamięci, dopóki istnieje blok.

Kolejna sprawa to kwestia deklaracji. Zmienna żyje (jest widoczna i dostępna w kodzie) dopiero od momentu deklaracji.

Przyjrzyjmy się przykładowi:

```
public class Main {
    public static void main(String[] args) {
        int a = 1;
        if (a < 10) {
            int b = 2;
            {
                int c = 3;
                System.out.println(a); // wydrukuje 1
                System.out.println(b); // wydrukuje 2
                System.out.println(c); // wydrukuje 3
            }
        }
    }
}
```

Ale to już nie zadziała:

```
public class Main {
    public static void main(String[] args) {
        int a = 1;
        if (a < 10) {
            int b = 2;
            {
                int c = 3;
            }
            System.out.println(a);
            System.out.println(b);
            System.out.println(c); // Błąd! nie ma dostępu (nie widzi) takiej zmiennej
        }
    }
}
```

Pamiętajmy też, że mimo innego zakresu nie możemy używać tych samych nazw w różnych blokach.

```
public class Main {
    public static void main(String[] args) {
        int a = 1;
        if (a < 10) {
            int a = 2; // błąd!
            {
                int a = 3; // błąd!
            }
        }
    }
}
```

O czym warto jeszcze pamiętać:

- zmienna zdefiniowana w bloku nazywana jest zmienną lokalną danego bloku.
- kiedy dany blok się wykona, dana zmienna przestaje istnieć (jest usuwana z pamięci)
- zasięg jest zagnieżdżony. Coś w zasięgu wyższym jest dostępne w zasięgu

zagnieżdżonym, ale nie odwrotnie.

```
{
    String name = "a";
    {
        System.out.println(name); // zmienna jest dostępna
    }
}
```

a tutaj przykład odwrotny:

```
{
    {
        String name = "a";
    }
    System.out.println(name); // zmienna nie jest dostępna
}
```