Java

Java

http://pl.wikibooks.org/wiki/Java
This Book Is Generated By Wb2PDF
using
RenderX XEP, XML to PDF XSL-FO Formatter

26-02-2013

Table of Contents

Java

Cechy Javy przemawiające za jej wyborem

- Przenośność - możesz uruchomić program na każdym systemie i sprzęcie, na który istnieje implementacja wirtualnej maszyny Java.
- Wygoda - podstawowe klasy i mechanizmy Java są zaimplementowane tak, że dostarczają programiście wygodnych w użyciu bibliotek, która w wielu przypadkach znacznie przyśpieszają tworzenie aplikacji.
- Szybkość tworzenia aplikacji - w Java aplikacje pisze się szybciej niż w niektórych innych językach programowania. Wszystko dzięki użytecznym klasom i mądrze zaplanowanym bibliotekom, narzędziom oraz IDE.
- Duże wsparcie ze strony twórców środowisk programowania - dzięki środowiskom IDE, takim jak NetBeans czy Eclipse, programy możesz tworzyć jeszcze szybciej i jeszcze wydajniej używając profesjonalnych narzędzi, które ułatwią tworzenie dużych aplikacji i panowanie nad tworzeniem oraz utrzymywaniem kodu.
- Ciekawe możliwości tworzenia interfejsu użytkownika - możliwość dostępu do wygodnych w użyciu bibliotek i narzędzi pozwalających szybko i bezboleśnie stworzyć niezależny od systemu i przenośny interfejs graficzny dla aplikacji. Biblioteki graficzne udostępniają metody, które są dokładnie tym, czego potrzebujesz.
- Duża ilość publikacji - w sieci jest sporo informacji na temat języka Java. Jest to ogromna zaleta szczególnie dla osób uczących się tego języka. W Internecie znaleźć można sporo przykładowego kodu, artykułów czy tutoriali.
- Garbage collection, czyli automatyczne zwalnianie nieużywanych już obszarów pamięci

Wady, o których warto pamiętać

- Przeważnie programy uruchamiane pod maszyną wirtualną działają wolniej niż programy napisane w C++ (szczególnie czasochłonne jest uruchamianie aplikacji).
- Użytkownik potrzebuje zainstalowanej maszyny wirtualnej javy (JVM), aby móc uruchomić nasz program.

- Java ustępuje innym językom programowania (np. C++ lub Ada) w tworzeniu aplikacji czasu rzeczywistego.

Czego potrzebujesz, żeby zacząć

Java to nie tylko język programowania - to również środowisko uruchomieniowe, w którym działają programy, tak zwana Wirtualna Maszyna Java (*Java Virtual Machine* - JVM). Java jest zaliczana do języków kompilowano-interpretowanych - aby napisane przez nas programy zadziałały, wymagany jest kompilator, który przekształci kod źródłowy do tzw. *byte code*, czyli odpowiednika kodu maszynowego rozumianego przez JVM (w tej książce funkcjonuje również termin *kod bajtowy*). Zarówno JVM jak i kompilator są dostarczane za darmo przez twórców Javy - firmę Oracle. Najlepiej ściągnąć wszystko w jednym pakiecie, który obecnie nosi nazwę **Java SE** (Java Platform, Standard Edition). Java SE jest dostępna na wiele platform, w tym dla systemów 32- i 64-bitowych Windows i GNU/Linux. Jest to wszystko, czego potrzebujemy do rozpoczęcia pracy.

Linux

Istnieje kilka implementacji Javy.[1] My wybierzemy otwartą implementację OpenJDK[1].

Debian/Ubuntu

W konsoli wpisujemy:

```
$ sudo apt-get install openjdk-6-jdk openjdk-6-jre openjdk-6-doc
```

Arch Linux

W konsoli wpisujemy (jako root):

```
# pacman -S jdk7-openjdk jre7-openjdk
```

lub jeśli potrzebujemy wersji 6:

```
# pacman -S openjdk6
```

1. ↑ Ubuntu Documentation > Community Documentation > Java

Następnie możemy sprawdzić działanie wpisując:

```
$ javac -version
```

Java z punktu widzenia programisty

Środowisko uruchomieniowe i developerskie

Aby dobrze programować, warto zrozumieć, jak wygląda całe podłoże mechanizmu Java. Pierwszym podstawowym elementem jest JRE (Java Runtime Environment - środowisko uruchomieniowe Javy). JRE jest niezbędne do uruchamiania aplikacji Java na komputerze. A co to oznacza dla nas programistów? Oznacza nie mniej, nie więcej, a dokładnie tyle, że programy pisane w tym języku uruchamiane są nie "w komputerze", ale w pewnym środowisku działającym na komputerze. Dzięki temu jesteśmy pewni, że jeśli powstanie implementacja Wirtualnej Maszyny Java (JRE - w naszym przypadku) na jakąkolwiek maszynę, to będziemy mogli na tej maszynie uruchomić nasz program bez względu na to, czy będzie to: Atari ST, XBOX czy najnowsza wersja tostera marki "dla Ciebie dla Domu". Dzięki temu programy pisane w Javie będą mogły być uruchamiane nawet na sprzęcie, który pojawi się za 10-20 lat, o ile będzie istniała implementacja Wirtualnej Maszyny Java dla tych urządzeń.

Wykonywanie aplikacji języka Java poprzez JRE rozwijało się w następujących kierunkach:

- Początkowo program mógł być interpretowany instrukcja po instrukcji (podobnie jak w językach skryptowych) jednak rozwiązanie to było bardzo niewydajne i powodowało, że programy napisane w języku Java działały bardzo wolno.
- Aby zwiększyć wydajność programów napisanych w języku Java w nowoczesnych maszynach wirtualnych Java, zaimplementowano mechanizmy mające przyśpieszyć działanie programów. Jednym z nich jest technika JIT (Just In Time). Technologia Just In Time kompiluje kod bezpośrednio przed jego wykonaniem. Dzięki temu kompilowane są tylko i wyłącznie funkcje czy klasy, których używamy w naszym programie (a nie cały program). JIT w połączeniu z optymalizacją adaptacyjną pozwala działać programom napisanych w Java niemal tak szybko jak aplikacjom C/C++.

Poza JRE (niezbędnym użytkownikowi) istnieje inna popularna forma dystrybucji pakietu Java. JDK (Java Development Kit), zwana również SDK (Software Development Kit). To dystrybucja języka Java dla osób tworzących kod w języku Java. Poza środowiskiem uruchomieniowym zawiera między innymi: kompilator, dokumentację/pomoc i debugger.

Kod wykonywalny programu

Po napisaniu kodu źródłowego program kompilowany jest do bytecodu. Nie jest to jeszcze kod zrozumiały dla procesora w sposób bezpośredni, który pozwalałby nam na jego uruchomienie. Jest to jednak kod zapisany w określonym formacie, który może zostać poprawnie zinterpretowany przez Maszynę Wirtualną Java, przetłumaczony na kod wykonywalny i uruchomiony.

Zwalnianie pamięci w języku Java - Garbage Collector

W języku Java w tle, podczas działania naszego programu, działa mechanizm zwany Garbage Collector (zbieracz śmieci). Ma on na celu zwolnienie programisty z obowiązku dbania o zwalnianie pamięci w programie. Stosuje on szereg algorytmów mających na celu wyłapanie niepotrzebnych obiektów i usunięcie ich. Posiada to swoje wady i zalety.

Zaletą jest to, że programista nie musi pamiętać o zniszczeniu obiektu lub zwolnieniu pamięci. Mamy wolny czas, który normalnie poświęcilibyśmy na tworzenie destruktorów i czasochłonne myślenie nad tym, czy oby na pewno dobrze alokuję i zwalniam pamięć. Więcej nawet - istnieją problemy w których nie jesteśmy w stanie rozstrzygnąć, czy obiekt powinien zostać "już" zwolniony czy jeszcze powinien pozostać przy życiu. GC zrzuca z nas również ten ciężar.

Wadą jest to, że działanie Garbage Collectora zajmuje czas. Nigdy nie wiadomo kiedy Garbage Collector postanowi zadziałać i wyszukać oraz zwrócić do systemu nieużywaną już przez program pamięć. Przez to systemy czasu rzeczywistego pisane w Javie obwarowane są dodatkowymi restrykcjami, ponieważ działanie programu może być w każdej chwili wstrzymane na odśmiecanie.

Najprostsze mechanizmy działania Garbage Collectora to usuwanie obiektów, dla których ilość referencji wynosi zero (z takiego obiektu nie skorzystamy bo nie mamy się do niego jak odwołać, więc nie ma sensu trzymać go w pamięci) i odnajdywanie trójkącików (referencja A wskazuje na B, B wskazuje na C ale C znowu wskazuje na A), ale istnieją również bardziej złożone.

Ciekawostki

Istnieją właściwie dwa GC. Jeden uruchamiany jest często i zwalnia zmienne które nie są już potrzebne, a które zostały utworzone "niedawno". Np. użyte zmienne lokalne po opuszczeniu metody która ich używała. Drugi jest uruchamiany rzadziej, ale czyści dokładniej całą pamięć, jednak jego działanie może powodować zatrzymanie JVM do czasu zakończenia procesu odśmiecania (istnieją jednak implementacje JVM które nie posiadają tej wady). Powinieneś wiedzieć, że GC w "zasadzie działa", ale dla aplikacji bardzo intensywnie używających pamięć może zdarzyć się sytuacja, że GC nie zdąży z odśmieceniem nim nastąpi przepełnienie pamięci - w takim przypadku zostanie zrzucony wyjątek OutOfMemoryError i aplikacja w zasadzie przestanie

działać. W takich programach warto używać przypisania do "null" - pomoże to GC łatwiej odnajdywać obiekty do usunięcia. Można również ręcznie zainicjować działanie GC poprzez użycie System.gc() - nie gwarantuje to, że wszystkie śmieci zostaną zwolnione, ale często może pomóc. Wgłębiając się w różne aspekty pamięci używanej przez JVM odnajdziesz obszar PermGen. Obszar PermGen nie jest odśmiecany (tak mówi doświadczenie) przez JVM, przechowuje się w nim między innymi informacje o klasach załadowanych przez ClassLoadera. Był taki problem z biblioteką javaassist, używanej przez kontener EJB3, który powodował ciągłe powiększanie obszaru zajmowanego PermGen, aż do katastrofy JVM. Do sterowania (w pewnym stopniu) GC mogą służyć parametry uruchomieniowe JVM.

Notka dla programistów C++

- W Java nie ma wielkości globalnych. Ponieważ w dobrym przybliżeniu wszystko jest klasą (dokładnie rzecz ujmując może być klasą), więc wartość globalna nie istnieje (każda wartość jest składową jakiejś klasy lub zmienną lokalną).
- Wszystkie klasy są pochodnymi od jednej wspólnej klasy Object.
- W Java nie ma wielokrotnego dziedziczenia. W Java występuje za to mechanizm interfejsów. Jedna klasa może implementować wiele interfejsów.
- Brak preprocesora. Nie jest potrzebny ponieważ wszystkim może zająć się Wirtualna Maszyna Java.
- Nie ma plików nagłówkowych. Program podzielony jest na pakiety.
- Brak typedef - tworzenie alternatywnych nazw dla istniejących typów nie jest wspierane.
- Brak możliwości przeciążania operatorów.
- Operator zakresu :: jest zastąpiony znakiem kropki.
- W języku Java nie istnieje możliwość przekazania domyślnej wartości, ani listy inicjalizującej w C++.
- Java posiada wsparcie dla wielowątkowości.
- Brak destruktorów i konieczności pamiętania o zwalnianiu pamięci. Działa garbage collector, który robi to za nas.
- Nie ma wskaźników - wszystko przekazywane jest przez referencje.
- W Javie łańcuchy znaków są obiektami klasy java.lang.String.
- W Javie argumenty typów prymitywnych są przekazywane do metod przez wartość.

Nauka programowania w języku Java

Naukę programowania można podzielić na pewne etapy. Pierwszy to poznanie struktur języka takich jak: pętle, warunki, operatory, podstawowe wyrażenia i składnia. Nie da się w tym czasie uniknąć osobnego działu, jakim jest korzystanie z bibliotek dostępnych w języku. Choćby po to, by wyświetlić coś na ekranie lub pobrać wiadomość od użytkownika. Te dwa elementy przeplatają się w początkowych etapach nauki języka. Później przychodzi czas na szczegóły związane z semantyką oraz składnią kodu, niuansami, dzięki którym można tworzyć optymalny kod i pisać kod sprytniej i wydajniej.

Poza nauką języka programowania trzeba poznać paradygmaty programowania jako takiego czyli przyjęte, obowiązujące lub zalecane reguły pisania programu, niezależne od tego w jakim języku programowania (wysokiego poziomu) piszemy. Jest to tak zwany dział "inżynierii oprogramowania", którym tutaj nie będziemy się zajmować.

Potraktuj naukę języka programowania jak etapy, w których jesteś wyposażany w kolejne narzędzia. Na początku Twoja wiedza jest równa niemalże zeru, później umiesz pierwszą rzecz, drugą, trzecią... jesteś wyposażany w kolejne narzędzia, poznajesz więcej mechanizmów, których możesz użyć do swojej pracy z językiem.

Komunikacja z użytkownikiem

Z racji struktury używanych dziś systemów operacyjnych istnieją dwie podstawowe warstwy komunikacji z użytkownikiem: konsola i GUI (Graphic User Interface). Pierwsza z nich to tak zwana konsola, czyli literki w okienku (czy na ekranie). Komunikowanie się z użytkownikiem (wyświetlanie informacji, pobieranie danych od użytkownika) jest znacznie prostsze w trybie konsoli. Dlatego też wszystkie początkowe składniki Javy poznamy pisząc programy działające właśnie pod konsolą. Unikniemy w ten sposób komplikowania kodu fragmentami charakterystycznymi dla bibliotek obsługujących graficzny tryb użytkownika, który poznamy w dalszej części kursu. Pozwoli to skupić się i wyeksponować to czego uczymy nie zaciemniając kodu niepotrzebnymi fragmentami, niezwiązanymi z tematem. Kiedy już oswoisz się z Javą jako taką - nic nie stoi na przeszkodzie abyś zaczął tworzyć aplikacje posiadające graficzny interfejs - jednak naukę najlepiej rozpocząć od prostych aplikacji pisanych pod konsolą.

Kurs podstawowy

W tej części postaram się zapoznać Cię z podstawowymi elementami języka takimi jak: pętle, warunki, zmienne itp. Na początku każdego zagadnienia będzie kilka słów teorii, później postaram się zobrazować go w postaci kodu programu. Do niektórych działów będę dodawał dodatkowe

paragrafy uściślające pewne kwestie, ważne dla osób już znających się na programowaniu. Jeżeli dopiero zaczynasz swoją przygodę z pisaniem programów, nie musisz ich czytać, są one tam raczej dla zaawansowanych programistów potrafiących już posługiwać się językiem programowania w celu zwrócenia uwagi na pewne kwestie mogące odgrywać rolę w ich programach - osobie dopiero uczącej się mogłyby tylko zaciemnić obraz całości i stać się niepotrzebną na początku komplikacją.

Ponieważ aby coś wyświetlić lub pobrać jakieś dane od użytkownika trzeba użyć bibliotek oraz w trakcie programowania używać pewnych konwencji - niektóre fragmenty mogą wydać się dla Ciebie na samym początku niezrozumiałe - a do ich objaśnienia dojdziemy później. Postaraj się nie zwracać na nie uwagi w początkowej fazie nauki, przyjąć, że "tak się robi" i poczekać trochę aż z czasem wszystko się wyjaśni. Zaczynamy!

Jak kompilować / uruchamiać programy

Jeżeli używasz konsoli najpierw musisz skompilować kod (przetłumaczyć na język zrozumiały dla Wirtualnej Maszyny Java). Robisz to poleceniem:

```
javac NazwaPliku.java
```

Powstanie wtedy plik **NazwaPliku.class**, który będzie już programem w języku Java. Aby go uruchomić wykonaj:

```
java NazwaPliku
```

i po problemie.

Jeżeli używać środowiska IDE nie będziesz musiał wpisywać żadnych komend. W Eclipse wystarczy, że wybierzesz: **Run \ Run As \ Java Application** . W NetBeans zaś wybierz **Run \ Run Main Project** (musisz tylko zwrócić uwagę który projekt jest wyróżniony po lewej stronie na liście **projects**). Jeżeli chcesz uruchomić inny projekt, kliknij na jego nazwę prawym przyciskiem myszy i wybierz **Set As Main Project**.

Struktura podstawowego programu

W Javie każdy program posiada pewną sztywną strukturę. Nie proszę, abyś ją zrozumiał (jeszcze nie teraz), ale żebyś po prostu zapamiętał ją i przyjął, że taka jest (z czasem wszystko stanie się dla Ciebie jasne).

```
public class NazwaPliku {
  public static void main (String [] args) {
```

```
  }
}
```

Jeżeli chciałbyś uruchomić ten program, musiałbyś go zapisać w pliku `NazwaPliku.java`. Po prostu w miejscu gdzie widnieje **NazwaPliku** musisz wpisać nazwę pliku, w którym zapisujesz swój program. Jeżeli plik będzie nazywał się `Pusty.java` to powyższy kod będzie wyglądał tak:

```
public class Pusty {
  public static void main (String [] args) {
  }
}
```

Dobrym nawykiem jest pisanie nazwy klasy wielką literą. Nazwa pliku z kodem musi być taka sama jak nazwa klasy. W tym przypadku to "Pusty", a plik, w jakim zapiszemy kod, to `Pusty.java`.

Powyższy program nic nie robi - jest jak łupinka orzecha pustego w środku. Jeżeli Twój edytor wspiera tworzenie szablonów nowych dokumentów, to dobrym pomysłem może okazać się stworzenie właśnie takie szablonu w edytorze - przynajmniej na początek. Zwolni Cię to z obowiązku przepisywania tej struktury za każdym razem.

Struktura programu dla zaawansowanych

Ponieważ w języku Java wszystko jest obiektem, również program musi być reprezentowany jako klasa. JRE proszone o uruchomienie pliku, wywołuje domyślnie metodę `main` klasy, o nazwie identycznej z nazwą pliku. Jeżeli w pliku nie znajduje się klasa o tej samej nazwie zostanie zwrócony błąd. Wszystko, co znajduje się w metodzie `main`, to właśnie nasz program. Ponieważ nigdzie nie jest tworzony obiekt naszej klasy, metoda `main` musi być statyczna, aby można było ją wywołać bez tworzenia instancji klasy (czyli bez tworzenia obiektu). Nakłada to na nas pewne ograniczenia - na przykład nie możemy korzystać z `this` (nie ma ono żadnego sensu w metodzie statycznej). Metodą pozwalającą ominąć ten problem jest stworzenie obiektu w metodzie `main` i przeniesienie kodu do konstruktora:

```
public class Pusty{

  public Pusty() {
    // Tutaj kod programu
  }

  public static void main(String [] args) {
    new Pusty();
  }

}
```

Powyższy kod tworzy obiekt klasy, którą projektujemy - dzięki temu ograniczenia związane z pisaniem kodu w metodzie statycznej znikają. Zastanawiać może jeszcze fragment kodu `String [] args` - jest to nic innego jak przekazanie do programu listy (tutaj tablicy) stringów (łańcuchów znaków czy jak wolisz napisów), będących parametrami wywołania Twojego programu. To właśnie poprzez zmienną `args` (arguments) otrzymujesz dostęp do wszystkich parametrów z jakimi została uruchomiona twoja aplikacja. Przekazanie parametrów od wersji *Java SE5* może wyglądać również tak:

```
 public static void main(String ... args) {
```

Zmienne

Zmienne możesz sobie wyobrazić jak pudełka, które najpierw opisujesz jakąś etykietką (nazwą), potem coś do takiego pudełka wkładasz i to tam trzymasz. Jest tylko jeden warunek: musisz wcześniej powiedzieć, co będziesz w tym pudełku trzymał. W analogii do zwierząt: inne pudełko weźmiesz dla psa (budę), inne dla rybki (akwarium). Dlatego musisz wziąć odpowiednie pudełko w zależności od tego, co chcesz trzymać. Potem możesz nazwać to pudełko. Znowu analogicznie możesz nakleić nalepkę z napisem "Nemo" na akwarium - informującą o tym, że rybka w akwarium nazywa się Nemo. Może być wiele rybek, ale twoją odróżnia od innych to, że nazywa się Nemo.

Jeżeli nie zrozumiałeś analogii ze zwierzątkami, to może porównanie matematyczne da Ci do myślenia. Kiedy rozwiązujesz jakieś zagadnienie matematyczne, gdzieś na końcu pojawia się coś w stylu:

```
x = 2
```

Cały czas posługiwałeś się symbolem **x** no i nagle okazuje się, że jest równy 2 ... jeżeli **x = 2** to

```
x + x + x = 6
```

zgadza się ? Czyli posługujesz się x-em jak dwójką i za każdym razem gdy piszesz x myślisz 2. Identycznie sprawa ma się ze zmiennymi.

Wróćmy do zmiennych jako takich. Zmienne to nic innego, jak nazywanie w programie napisów i liczb. Możesz liczbie 2 przypisać dowolną nazwę. Używane nazwy mogą być niemalże dowolnie długie (nie ważne czy zmienna będzie nazywała się **x** czy **niewiadoma_rownania_x**). W nazwach najlepiej ograniczać się do stosowania małych i wielkich liter, cyfr i znaku podkreślenia.

Typy zmiennych

Podstawowe informacje, jakie możemy chcieć przetrzymać w komputerze, to: wartość logiczna, liczba, napis lub pojedynczy znak. Mamy więc różne **typy** zmiennych. Podstawową wartością, którą możesz chcieć przechować sobie gdzieś (zanotować na tak długo jak Ci jest potrzebna - a tak działają zmienne), jest liczba. Komputer rozróżnia liczby całkowite (2; -15; 123) i zmiennoprzecinkowe (2,14; 3,14; -1,5). Do trzymania liczby **całkowitej** używać będziemy typu **int**. Tak więc napisanie:

```
int liczba;
```

stworzy zmienną o nazwie liczba. Teraz powrót do analogii pudełek. Stworzyłeś pudełko, które umie trzymać liczby całkowite i nazwałeś je **liczba**. Masz więc pudełko **liczba**. Ale co w nim jest? Nie wiadomo - i naprawdę to jest poprawna odpowiedź. Musisz pamiętać! Jak tworzysz pudełko to ono w ogólności na starcie nie jest puste (tylko w pewnych szczególnych przypadkach). Aby teraz na przykład włożyć do tego pudełka liczbę 2 piszesz:

```
liczba = 2;
```

no i w pudełku znalazła się liczba 2, a to co było w środku, zostało automatycznie wyrzucone. Można w jednej linijce stworzyć pudełko i włożyć do niego coś pisząc:

```
int liczba = 2;
```

Teraz wartości zmiennoprzecinkowe. Typ zmiennej nazywa się **float**:

```
float liczba_zmiennoprzecinkowa = 3.14;
```

stworzy pudełko do trzymania liczb zmiennoprzecinkowych (**float**) o nazwie **liczba_zmiennoprzecinkowa** i włoży do tego pudełka wartość **3,14**. Teraz przechodzimy z terminologii pudełkowej do terminologii zmiennych - najwyższy czas! Więc teraz uważaj. Do zapisania pojedynczego znaku użyjesz *typu* **char**.

```
char znak = 'a';
```

Powyższa linijka *utworzy* zmienną *typu* **char** o *nazwie* **znak** i *wartości* **a** (zawartość pudełka będziemy nazywać wartością zmiennej). Zapewne trapi Cię cały czas pytanie: "skąd tam się wziął średnik na końcu". Otóż programowanie polega na wydawaniu komputerowi poleceń. Tak jak to powyższe "Utwórz zmienną typu znakowego o wartości **a**". Każde takie polecenie w języku Java musi kończyć się średnikiem. Programując przyjmuje się, że zapisujemy jedno polecenie w jednej linijce (choć można w jednej linijce zapisać więcej poleceń - są wtedy po prostu oddzielone średnikami). Dzięki temu program wygląda tak, że **każda linijka kończy się średnikiem** -

choć tak naprawdę **każde polecenie (instrukcja) kończy się średnikiem**. Ot cała filozofia średników. Zwróć jeszcze uwagę, na to, że pojedyncze znaki zapisujemy w apostrofach. Użycie cudzysłowu byłoby błędem. Powyższą wiedzę przedstawię w tabelce:

Typ zmiennej	Przeznaczenie	Przykład	Uwagi
int	liczby całkowite	`int liczba = -777;`	brak
float	liczby zmiennoprzecinkowe	`float pi = 3.14;`	brak
char	pojedynczy znak	`char znak = 'A';`	zawsze zapisuje się **pomiędzy apostrofami**

Każdą** instrukcję kończ znakiem średnika **;

Pierwszy program

Utwórzmy plik `Hello.java` o poniższej treści.

```
//główna klasa programu:
public class Hello {
  //główna metoda programu:
  public static void main(String args[]){
    System.out.print("Witaj swiecie!"); //ten tekst w nawiasie zostanie wyświe-
tlony:
  }
}
```

Najlepiej do tego celu użyć edytora plików tekstowych kolorującego składnię. Dzięki temu bez problemu odróżnisz treść programu od licznych komentarzy, których obecność ma pomóc zorientować się w temacie i nie wpływa w żaden sposób na działanie naszego programu.

Ja użyłem darmowego programu Vim (wersję dla twojego systemu operacyjnego pobierzesz spod adresu http://www.vim.org/download.php. Możesz skopiować powyższe źródło programu, lub przepisać je ręcznie (w celu rozpoczęcia ręcznej edycji tekstu programu, należy przestawić Vima w tryb edycji za pomocą klawisza **a**), a następnie zapisać w pliku *Hello.java* .

Ponieważ domyślnie Vim koloruje składnię języka programowania na podstawie rozszerzenia otwieranego pliku, dopiero po ponownym otwarciu pliku `Hello.java` zobaczymy kolorową składnię języka Java.

Chcąc skompilować nasz program do postaci kodu bajtowego, wydajemy polecenie `javac Hello.java`. Utworzony zostanie wówczas plik o nazwie `Hello.class`, który - ponieważ zdefiniowaliśmy metodę główną `main( )` - możemy wykonać za pomocą polecenia `java Hello`.

Obiektowe podstawy

Podobnie do pierwszego programu napisanego w języku *Java* , aby uruchomić poniższe przykłady, należy je wcześniej skompilować do kodu bajtowego za pomocą polecenia **javac <nazwa pliku o rozszerzeniu .java>**, a następnie uruchomić na Wirtualnej Maszynie Javy za pomocą polecenia **java <nazwa pliku o rozszerzeniu .class pisana tutaj bez rozszerzenia>**. Pamiętajmy, że uruchomić możemy tylko tę klasę, która zawiera metodę *main( )*. Zatem dla *Przykładu 1* po wydaniu polecenia **javac Proba.java**, w celu wykonania programu należy wykonać polecenie **java Proba**.

```
T:\archiwum\java>java Proba
Jan Kowalski, 1981, PESEL: 81111224350

T:\archiwum\java>
```

Wykonanie polecenia `java Proba.class` wygeneruje wyjątek podobny do tego poniżej. Polecenie `java` jako argumentu oczekuje nazwy klasy, a nie nazwy pliku, do którego skompilowano klasę. Stąd poniższy komunikat.

```
T:\archiwum\java>java Proba.class
Exception in thread "main" java.lang.NoClassDefFoundError: Proba/class

T:\archiwum\java>
```

Również polecenie `java Person` zgłosi wyjątek:

```
T:\archiwum\java>java Person
Exception in thread "main" java.lang.NoSuchMethodError: main

T:\archiwum\java>
```

ponieważ klasa `Person` nie zawiera metody `main( )`, której deklaracja jest konieczna, jeśli chcemy uruchamiać nasz program z wiersza poleceń. W naszym przykładzie klasa `Person` zawiera wyłącznie definicje typów danych, składających się na opis osoby.

Przykład 1

Listing pliku `Proba.java` :

```
public class Proba {
public static void main(String[] args) {
                Person p, q, r;
                p = new Person();
                q = new Person();
                r = new Person();

                p.firstname = "Jan";
                p.lastname = "Kowalski";
                p.year = 1981;
                p.PESEL = "81111224350";

                q.firstname = "Anna";
                q.lastname = "Nowak";
                q.year = 1975;
                q.PESEL = "75032074926";

                System.out.println( p.firstname + " " + p.lastname + ", " +
                                    p.year + ", PESEL: " + p.PESEL );

                System.out.println( q.firstname + " " + q.lastname + ", " +
                                    q.year + ", PESEL: " + q.PESEL );

        }
}

class Person
{
        public String firstname, lastname;
        public int year;
        public String PESEL;
}
```

Plik zawiera definicje dwóch klas *Person* oraz *Proba* . Pierwsza z nich definiuje typ w postaci, w którym będziemy przechowywać dane na temat przykładowych osób *Jana Kowalskiego* i *Anny Nowak*. Klasa *Proba* zawiera przykład zastosowania klasy *Person* w krótkim programie wpisującym dane osobowe naszych bohaterów *Jana* i *Anny* oraz wyświetlającym je w zrozumiałej dla użytkownika formie. Jak widać z przykładu, pomimo tego, że wzorzec jest wspólny dla trzech obiektów (definiacja klasy Person), każdy z nich przechowuje swoje niezależne dane. Powyższy kod jest **jedynie przykładem** i zastosowano w nim konwencje, których w normalnym programowaniu nie powinno się stosować (np. publiczne atrybuty obiektu).

Przykład 2

Najlepiej w nowo utworzonym folderze edytujemy plik o nazwie *Test.java* wg poniższej treści:

```
class Complex
{
        public double re, im;           // odpowiednik definicji rekordu
                                        // re - część rzeczywista liczby ze-
spolonej
                                      // im - część urojona liczby zespolonej


        public double mod()             // definicja metody w klasie
        {
          return Math.sqrt( this.re*this.re + this.im*this.im );
        }       // this oznacza dany obiekt w klasie Test bedzie to z.re i
z.im

        public String toString()
        {
          return "(" + this.re + ", " + this.im + ")" ;
        }


        public Complex coupled()
        {
          Complex other;        // bufor dla this, żeby nie zmienił
          other = new Complex();        // swojej wartości .im

//!       other = this; // błędne, ustawienie 'this' do zmiennej
          other.re = this.re; // ustawiamy tylko wartości
          other.im = this.im;

          other.im = -other.im;
          return other;
        }
}

class Test
{
    public static void main(String[] args) {
          Complex x, y, z;      // x,y,z reprezentują obiekty klasy Complex
          x = new Complex();    // tworzenie nowego obiekt "x"
          y = new Complex();
          z = new Complex();

          x.re = 5.0;  x.im = 3.0;
          y.re = 4.0;  y.im = 4.0;
```

```
          System.out.print
          ( "Liczby zesp.: " + x.toString() );  // do liczby mozna dodac
                // nie moze byc przecinkow      // string, wowczas liczba
                // w tym nawiasie               // staje się stringiem

          System.out.println( ", " + y.toString() );
          System.out.println( "========================================" );


          z.re  =  x.re + y.re;
          z.im  =  x.im + y.im;
          System.out.println( "suma liczb: " + z.toString() );

          z.re  =  x.re*y.re - x.im*y.im;
          z.im  =  x.re*y.im + x.im*y.re;
          System.out.println( "iloczyn liczb: " + z.toString() );

          System.out.println( "modul pierwszej: " + x.mod() );

          z = x.coupled();      // tworzy sprzężoną do x i wstawia ja
                                // w miejsce z

          System.out.println( x.toString() + "sprzezona do pierwszej: " +
z.toString() );

    }
}
```

Mamy tutaj zdefiniowane dwie klasy *Complex* i *Test*. Zatem po zakończeniu kompilacji do kodu bajtowego otrzymamy dwa pliki z rozszerzeniem *.class* . Klasa *Complex* zawiera definicje typu liczby zespolonej, na przykładzie którego wykonywać będziemy operacje dodawania, mnożenia, itd. przykładowych liczb zespolonych. W klasie *Test* znajdują się polecenia wykorzystujące typ i metody zdefiniowane dla klasy *Complex*.

Przykład 3

Plik źródłowy *CrashTest2.java* :

```
class Vehicle
{
        private String owner;
        public String model;
        public int type; //1,2,3 - typy samochodow
        private int reg;

        public void setOwner(String s) { owner=s; }
        public String getOwner() { return owner; }

//      public String toString()
```

```
//      { return model + ", " + type + "\nReg.No." + reg +
//                        ", wlasnosc: " + owner + "\n"; }

        public String toString()
        { return model + ", " + typeToString() + "\nNr " + reg +
                                                ", wlasnosc: " + owner + "\n";
 }

        private static int count=1000; /* poniewaz jest static, z kazdego
                                          obiektu nastepuje odwolanie do tej
                                          samej komorki pamieci */

        Vehicle() { reg = count++; } // konstruktor klasy

        private String typeToString()
        {
           if (type == 1) return "rower";
           else if (type == 2) return "motocykl";
           else if (type == 3) return "auto";
           else return "NIEZNANY!";
        }
}

class CrashTest2
{
        public static void main(String[] args)
        {
           Vehicle v1 = new Vehicle();
           Vehicle v2 = new Vehicle();
           Vehicle v3 = new Vehicle();

           v1.model = "Syrena 105 Turbo";
           v1.type = 3;
           v1.setOwner("Jan Kowalski");
           System.out.println(v1);

           v2.model = "Harley D., 1965";
           v2.type = 2;
           v2.setOwner("Crazy Jackill");
           System.out.println(v2);

           v3.model = "Romet Wigry 3";
           v3.type = 1;
           v3.setOwner("John Brown");
           System.out.println(v3);
        }

}
```

Przykład 4

Plik o nazwie *CrashTest.java* :

```
class Car
{
        public String make;

        public String model;

        public int year;

        private String reg;

        public String toString()
        { return make + ", " + model + ", " + year + ", " + reg; }

/* jesli cos jest prywatne, to dostep do tej rzeczy moze sie odbyc
poprzez jakis wewnetrzny mechanizm, w tym wypadku beda to ponizsze
metody */

        public String getReg()
        { return reg; }

/*      public void setReg(String r)
        { reg = r; }
 Zamiast powyzszej metody wprowadzamy bardziej zaawansowana.
*/

        public void setReg(String r)
        {       if ( r.length() != 7 )
                {       /* System.out.println( "Bad reg:" + r );
                        System.exit(1); */
                        throw new IllegalArgumentException("Bad reg: " + r);
 // wyrzucamy wyjątek
                }
                reg = r;
        }


/* ponizej znajduja sie konstruktory wykorzystywane przy okazji
korzystania z klasy Car. Zasada polega na tym:
jesli deklarujemy choc jeden konstruktor, musimy zadeklarowac wszystkie
wykorzystywane. Do tej pory jedynym wykorzystywanym byl najprostszy
"p=new Car()". Drugim konstruktorem jest ten, za pomoca ktorego nadajemy
wartosc "q" */

        Car() { }

        Car(String mk, String md, int y, String rg)
        {       make = mk;
```

```
                model = md;
                year = y;
                setReg(rg);
        }
}



class CrashTest
{
        public static void
        main(String[] args)
        {
                Car p,q,r;
                p=new Car();
                p.make = "Wolga";
                p.model = "Cziornaja";
                p.year = 1968;

                //p.reg="SEB1999";
                p.setReg("SEB2000");

                q = new Car( "Fiat", "126p 4wd", 1972, "XXXyyyy" );
                //    === - wystepuje jako konstruktor


                System.out.println(p);

                System.out.println(q);
        }
}
```

Przykład 5 - prosta grafika

Plik o nazwie *Rysik.java* :

```
import java.awt.*;
import javax.swing.*;

public class Rysik
{
      public static void main(String[] args)
      {
            JFrame okno = new JFrame("Okno");
            okno.add(new Plansza());

            okno.setSize(100,100);

            // dzięki tej linii program zakończy się po zamknięciu okna
            okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
            /*
            *  bez poniższej linii kodu nasze okno będzie niewidoczne
            *  ustawienie setVisible(false) ukrywa okno, lecz go nie niszczy
            *  to znaczy, że możemy ukryć okno, a za chwilę je pokazać
            *  bez konieczności ponownego tworzenia go od podstaw
            */
            okno.setVisible(true);
      }
}


class Plansza extends JPanel
{
      Plansza()
      {
            // te informacje wydrukowane zostaną w konsoli/terminalu, a nie
w oknie programu!!
            System.out.println("Szerokość planszy:" + this.getWidth());
            System.out.println("Wysokość planszy:" + this.getHeight());
      }


      /*
      *  w tej funkcji umieszczamy kod ze wszystkim, co chcemy narysować
      *  jest to funkcja, która wywoływana jest automatycznie przez Javę
      *  za każdym razem, gdy zachodzi taka potrzeba (np. zmiana wielkości
      *  okna przez użytkownika); lepiej nie wywoływać jej na własną rękę
      */
      public void paint(Graphics g)
      {

            /*
            *  umieszczenie tych zmiennych jako zmiennych lokalnych
            *  funkcji paint() zapewni aktualizację tych zmiennych
            *  podczas zmiany wielkości okna przez użytkownika
            */
            int width = this.getWidth();
            int height = this.getHeight();

          // obiekt graficzny g rysuje linię po przekątnej panelu z marginesem
 10 pikseli
            g.drawLine(10, 10, width - 10, height - 10);
      }

}
```

Narzędzia

Edytory

- jEdit [1]
- Notepad++ [2]

IDE

- Eclipse [3]

Eclipse jest darmową platformą przeznaczoną do tworzenia desktopowych aplikacji Java, zaprojektowana początkowo przez firmę IBM, a następnie udostępniona i rozwijana na zasadach Open Source przez Fundację Eclipse (ang. Eclipse Foundation). Jest to tzw. *Rich Client Platform* – aplikacje tworzone na jej podstawie mogą posiadać wbudowaną logikę, w przeciwieństwie do tzw. *Thin Client Platform*, gdzie po stronie użytkownika dostępny jest jedynie interfejs aplikacji, natomiast wszystkie operacje wykonywane są na serwerze (przykładem takiego rozwiązania jest zasada działania przeglądarek WWW).

Eclipse dostępne jest dla wszystkich platform, które posiadają własną implementację wirtualnej maszyny Java oraz dla których przygotowano implementację opracowanej przez Fundację biblioteki graficznej SWT, będącej alternatywą dla standardowych bibliotek graficznych Javy – AWT i Swing. Sama platforma nie dostarcza żadnych narzędzi służących do tworzenia kodu i budowania aplikacji, oferuje jednak obsługę wtyczek rozszerzających jej funkcjonalność, umożliwiających m.in. rozwijanie aplikacji w językach Java, C/C++, PHP, tworzenie GUI, modelowanie UML, współprace z serwerami aplikacji, serwerami baz danych itp.

- Geany [4]

Geany jest to edytor tekstu stworzony z użyciem GTK2. Zawiera wszystkie podstawowe narzędzia z zintegrowanego środowiska programistycznego. Został on opracowany w celu zapewnienia małego i szybkiego IDE, które ma tylko kilka zależności od innych pakietów. Wspiera wiele typów plików, języków programowania oraz posiada kilka ciekawych funkcji.

- NetBeans [5]

NetBeans to kolejna platforma programistyczna, która ma na celu przyśpieszenie procesu tworzenia aplikacji pisanych w języku Java. Podstawowe mechanizmy wspierania programisty w pisaniu kodu to między innymi: podświetlanie błędów, pomoc w uzupełnianiu składni, automatyczne importowanie potrzebnych modułów czy zaznaczaniu niepotrzebnych fragmentów

kodu oraz wygodny i użyteczny debugger. Środowisko działa zarówno pod kontrolą systemu Linux, jak i Windows, dzięki czemu możesz bezboleśnie przesiadać się pomiędzy systemami, nadal pracując z tą samą platformą.

NetBeans wspiera również tworzenie aplikacji C/C++ oraz Ruby (RubyOnRails, po małych zmianach również Merb). Środowisko domyślnie implementuje wsparcie dla biblioteki Swing, między innymi graficzny designer do projektowania wyglądu aplikacji (okienek, kontrolek).

- JCreator [6] proste i szybkie IDE, dostępne tylko na platformę Windows.

Co wybrać ?

Na to pytanie musisz odpowiedzieć sobie sam. Prawdopodobnie najlepszym wyjściem jest dla Ciebie zainstalowanie wszystkich dostępnych w Twoim przypadku środowisk, stworzenie aplikacji, dwóch w każdym z nich oraz samodzielny wybór. Poeksperymentuj, sprawdź, które z narzędzi bardziej odpowiada Twoim wymaganiom, w którym przyjemniej i milej Ci się programuje, które bardziej przypadnie Ci do gustu.

Być może zaczniesz używać obydwu narzędzi w zależności od tego jaki projekt i z użyciem jakich bibliotek będziesz tworzył. Powodzenia!

Bibliografia

1. ↑ Ubuntu Documentation > Community Documentation > Java
2. ↑ Strona główna Openjdk, otwartej implementacji Javy

Warto przeczytać

- *Wolne, lecz w okowach - pułapka Javy* - artykuł Richarda Stallmana z dn.2004-04-12
- *95% wolnej Javy* - artykuł na łamach serwisu linuxnews.pl z dn. 2005-11-04
- *Wolna Java od Suna!* - artykuł na łamach serwisu linuxnews.pl z dn. 2006-11-13

Linki zewnętrzne

- http://java.sun.com/ - oficjalna witryna Suna dotycząca języka i środowiska Java
- http://math.hws.edu/javanotes/ - bardzo dobry kurs programowania w Javie od podstaw

- http://java.oz.pl/ - kurs programowania w Javie - po polsku i od podstaw
- http://arturt.republika.pl/java/ - kurs programowania w Javie po polsku
- http://4programmers.net/Java/Podstawy_Javy - podstawowy kurs java po polsku
- Java. Obiekty refleksyjne - artykuł wyjaśniający co to są refleksje i jak się je stosuje w Javie
- http://jdn.pl/ - Java Developers Network - portal programistów
- Kurs programowania obiektowego w Javie - Kurs z Uniwersytetu Warszawskiego
- Zaawansowany kurs programowania obiektowego w Javie - Druga część kursu z Uniwersytetu Warszawskiego
- Full Java Tutorial