Jeśli chcesz poznać Javę, nie szukaj dalej – oto pierwsza książka techniczna z graficznym interfejsem użytkownika!

Wydanie II. Obejmuje Javę 5.0

Rusz głową!

Java

Dowiedz się, jak wątki mogą poprawić Twoje życie

Uniknij zawstydzających błędów obiektowych

Sprawdź swój umysł na 42 zagadkach

Pomóż mózgowi utrwalić zagadnienia związane z Javą

Poigraj w bibliotece Javy

Twórz atrakcyjne i użyteczne interfejsy graficzne aplikacji

O'REILLY®

Kathy Sierra, Bert Bates

Helion

Java. Rusz głową! Wydanie II

Autor: Kathy Sierra, Bert Bates
Tłumaczenie: Piotr Rajca
ISBN: 978-83-246-2773-8
Tytuł oryginału: Head First Java, 2nd Edition
Format: 200×230, stron: 720

**Jeśli chcesz poznać Javę, nie szukaj dalej –
oto pierwsza książka techniczna z graficznym interfejsem użytkownika!**

Technologie oparte na Javie są wszędzie – jeśli piszesz oprogramowanie i jeszcze nie poznałeś Javy, nadszedł czas, by ruszyć głową!

Otwórz się na Javę i projektowanie obiektowe. Przystąp do nauki unikalną metodą, wykraczającą poza suche opisy składni oraz sposobów omijania codziennie spotykanych raf programistycznych. To doskonały podręcznik dla osób, które lubią uczyć się nowych języków programowania i nie mają wykształcenia informatycznego lub programistycznego. Zostań fantastycznym programistą i zdobądź wiedzę kompletną: od tworzenia obiektów, po graficzny interfejs użytkownika (GUI), obsługę wyjątków (błędów) oraz komunikację sieciową (gniazda) i wielowątkowość, a nawet pakowanie grupy plików klasowych w jeden plik instalacyjny.

Naprzód, głowo! Nikt ci tego nie potrafił wytłumaczyć? Wydaje Ci się, że to problem nie na Twoją głowę? Nie potrzebujesz elektrowstrząsów, żeby pobudzić swój mózg do aktywnego działania. Tylko żadnych gwałtownych gestów! Usiądź wygodnie, otwórz książkę, dopiero teraz się zacznie. Na początek – rusz głową!

Precz z nudnymi wykładami i zakuwaniem bez zrozumienia! Nauka to znacznie więcej niż tylko czytanie suchego tekstu. Twój mózg jest niczym głodny rekin, cały czas pracy naprzód w poszukiwaniu nowej, apetycznej przekąski. Jak karmimy Twój wygłodniały umysł?

Używamy rysunków, bo obraz wart jest 1024 słów. Stosujemy powtórzenia, by zakodować na stałe dane w Twojej chłonnej głowie. Oddziałujemy na emocje, jesteśmy nieprzewidywalni, zaskakujący i zabawni. Stawiamy przed Tobą wyzwania i zadajemy pytania, które angażują Cię w proces studiowania przedstawianych zagadnień. Cały czas pobudzamy Twój umysł do aktywnego działania, zmuszamy go do posłuszeństwa… a za ciężką pracę nagrodzimy go smakowitym ciasteczkiem w postaci wiedzy – wisienka gratis!

Rozgryź to sam!

- Klasy i obiekty
- Typy danych
- Pierwszy program w Javie
- Java API
- Programowanie obiektowe – dziedziczenie, polimorfizm, interfejsy i klasy abstrakcyjne
- Metody
- Obsługa wyjątków
- Graficzny interfejs użytkownika
- Operacje wejścia-wyjścia
- Programowanie sieciowe i RMI

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl

Spis treści (skrócony)

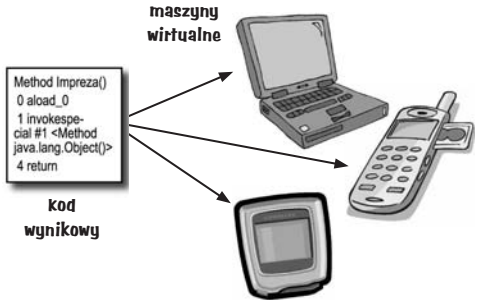
Spis treści (z prawdziwego zdarzenia)

Wprowadzenie

Twój mózg myśli o Javie. W tym rozdziale *Ty* próbujesz się czegoś dowiedzieć, natomiast Twój *mózg* robi Ci uprzejmość i stara się, aby te informacje nie zostały zapamiętane *na długo*. Myśli sobie: „Lepiej zostawić miejsce na ważniejsze rzeczy, takie jak dzikie zwierzęta, których należy się wystrzegać, lub rozważania, czy jeżdżenie na snowboardzie w stroju Adama to dobry pomysł". A zatem jak *można* oszukać własny mózg i przekonać go, że od znajomości Javy zależy nasze życie?

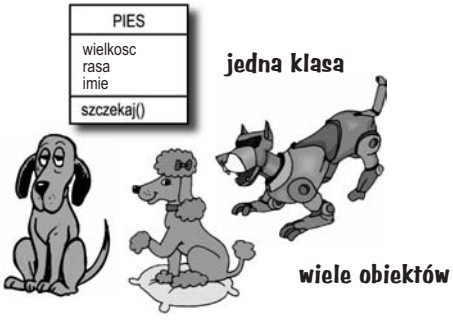
1 Przełamując zalew początkowych trudności

Java zabiera nas w nowe miejsca. Od momentu pojawienia się pierwszej, skromnej wersji o numerze 1.02, Java pociągała programistów ze względu na przyjazną składnię, cechy obiektowe, zarządzanie pamięcią, a przede wszystkim obietnicę przenośności. Od samego początku „wskoczymy na głęboką wodę" — napiszemy prosty program, skompilujemy go i wykonamy. Porozmawiamy o składni, pętlach, rozgałęzieniach oraz innych czynnikach, które sprawiają, że Java jest taka fajna. Zatem, wskakuj!



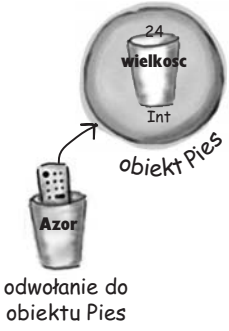
2 Wycieczka do Obiektowa

Mówiono mi, że będą tam same obiekty. W rozdziale 1. cały kod programu został umieszczony w metodzie `main()`. Nie jest to w pełni zgodne z zasadami programowania obiektowego. A zatem trzeba porzucić świat programowania proceduralnego i rozpocząć tworzenie własnych obiektów. Zobaczymy, co sprawia, że programowanie obiektowe w Javie jest takie fajne. Wyjaśnimy także, na czym polega różnica pomiędzy klasą a obiektem. W końcu pokażemy, w jaki sposób obiekty mogą ułatwić nam życie.



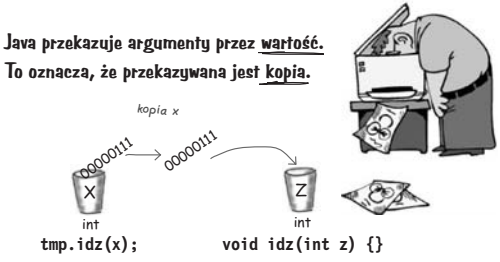
3 Poznaj swoje zmienne

Istnieją dwa rodzaje zmiennych: zmienne typów podstawowych oraz odwołania. W programach będą istnieć także inne typy danych niż jedynie liczby całkowite, łańcuchy znaków i tablice. Co zrobić, jeśli chcielibyśmy stworzyć obiekt **`WłaścicielZwierzaka`** zawierający składową **`Pies`**? Albo **`Pojazd`** ze składową **`Silnik`**? W tym rozdziale ujawnimy tajemnice typów danych w Javie oraz pokażemy, co można *zadeklarować* jako zmienną, *zapisać* w zmiennej oraz co z taką zmienną można potem zrobić. W końcu przekonamy się, jak wygląda życie na automatycznie porządkowanej stercie.



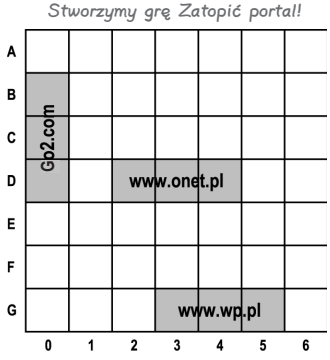
4 Jak działają obiekty?

Stan wpływa na działanie, a działanie wpływa na stan. Wiemy, że obiekty mają swój **stan** i określone **działanie** reprezentowane odpowiednio przez **składowe** i **metody**. Teraz sprawdzimy, w jaki sposób stan i działanie obiektów są ze sobą *powiązane*. Otóż w swych działaniach obiekty wykorzystują swój unikalny stan. Innymi słowy, ***metody wykorzystują wartości składowych obiektów***. Na przykład: „Jeśli pies waży mniej niż 20 kilogramów, szczekamy radośnie, w przeciwnym przypadku...”. ***Spróbujmy zmienić stan obiektu!***



5 Supermocne metody

Dodajmy naszym metodom nieco siły. Zajmowaliśmy się już zmiennymi, eksperymentowaliśmy z kilkoma obiektami i napisaliśmy kod paru programów. Jednak potrzeba nam więcej narzędzi. Na przykład **operatorów**. I **pętli**. Może przydałoby się **wygenerować kilka liczb losowych** i **zamienić łańcuch znaków na liczbę całkowitą**. O tak, to byłoby super. I dlaczego nie nauczymy się tego wszystkiego, *tworząc* coś rzeczywistego, by przekonać się, jak wygląda pisanie (i testowanie) programu w Javie od samego początku do końca. **Może jakąś grę**, taką jak „Zatopić portal" (podobną do gry w okręty).



6 Korzystanie z biblioteki Javy

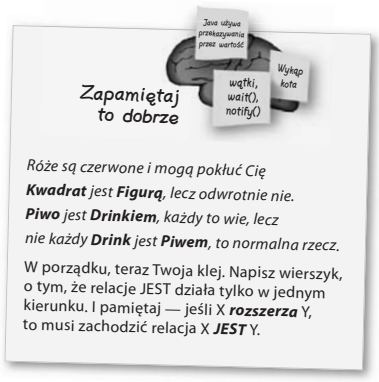
Java jest wyposażona w setki gotowych klas. Jeśli tylko dowiesz się, jak znaleźć w bibliotece Javy (nazywanej także **Java API**) to, czego szukasz, nie będziesz musiał ponownie wymyślać koła. *W końcu masz ciekawsze rzeczy do roboty.* Jeśli masz zamiar napisać program, to równie dobrze możesz napisać *tylko* te jego fragmenty, które są unikalne. Standardowa biblioteka Javy to gigantyczny zbiór klas, które tylko czekają, aby użyć ich jako klocków przy tworzeniu programów.

„Dobrze jest wiedzieć, że w pakiecie java.util istnieje klasa ArrayList. Ale jak mogłabym się o tym sama dowiedzieć?"

— Julia, lat 31, modelka

7 Wygodniejsze życie w Obiektowie

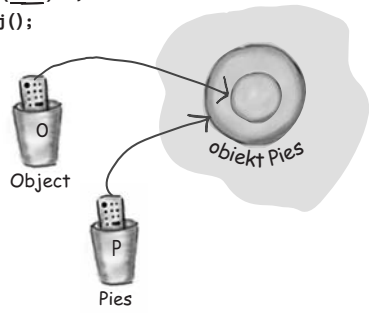
Planuj swoje programy, myśląc o przyszłości. A gdyby tak można tworzyć kod, który *inni* programiści mogliby rozbudowywać, **i to w prosty sposób**? A gdyby tak można było tworzyć elastyczny kod z myślą o tych najnowszych zmianach wprowadzanych w specyfikacji? Kiedy dowiesz się o Planie polimorficznym, poznasz 5 kroków służących projektowaniu lepszych klas, 3 sztuczki z polimorfizmem oraz 8 sposobów tworzenia elastycznego kodu, a jeśli zaczniesz już teraz, dodatkowo otrzymasz 4 podpowiedzi odnośnie wykorzystywania dziedziczenia.



8 Poważny polimorfizm

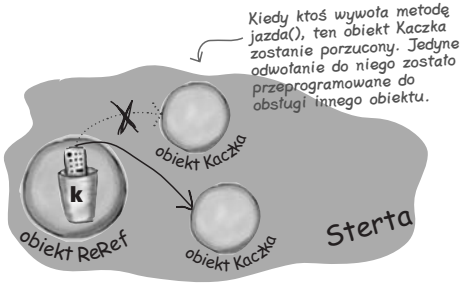
Dziedziczenie to tylko początek. Aby w pełni wykorzystać polimorfizm, niezbędne nam będą interfejsy. Musimy pójść dalej, poza proste dziedziczenie, i uzyskać elastyczność, jaką może zapewnić wyłącznie projektowanie i kodowanie z wykorzystaniem interfejsów. Czym jest interfejs? W 100% abstrakcyjną klasą. A czym jest klasa abstrakcyjna? To klasa, która nie pozwala na tworzenie obiektów. A co nam po takiej klasie? Przeczytaj, to się dowiesz...

```
Object o = lista.get(indeks);
Pies p = (Pies) o;
p.szczekaj();
```



9 Życie i śmierć obiektu

Obiekty się rodzą i umierają. To Ty jesteś za to odpowiedzialny. To Ty decydujesz, kiedy i jak *skonstruować* obiekt. Ty także decydujesz, kiedy go *porzucić*. **Odśmiecacz pamięci** odzyska pamięć zajmowaną przez niepotrzebne obiekty. Przyjrzymy się, w jaki sposób obiekty są tworzone, gdzie „żyją" i w jaki sposób efektywnie je przechowywać lub porzucać. Oznacza to, że będziemy dyskutować o stercie, stosie, zasięgach, konstruktorach, konstruktorach nadrzędnych, odwołaniach pustych oraz przydatności odśmiecacza pamięci.



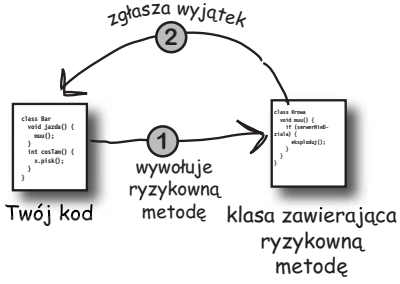
W składowej „k" jest zapisywany nowy obiekt Kaczka, przez co oryginalny (pierwszy) obiekt zostaje porzucony. Teraz ten pierwszy obiekt do niczego się nam już nie przyda.

10 Liczby mają znaczenie

Zabaw się w matematyka. Java API udostępnia metody do wyznaczania wartości bezwzględnej, zaokrąglania liczb, określania wartości minimalnej i maksymalnej i tak dalej. A co z formatowaniem? Moglibyśmy chcieć wyświetlać liczby ze znakiem dolara na początku i dwoma miejscami dziesiętnymi. Albo wyświetlić daną w formie daty. Albo daty w zapisie stosowanym w Anglii. A co z przekształcaniem łańcucha znaków na liczbę? Lub zamianą liczby na łańcuch znaków? Zaczniemy jednak od wyjaśnienia, co oznacza, że zmienne lub właściwości są *statyczne*.

11 Ryzykowne działania

Czasami zdarzają się nieprzewidziane sytuacje. Pliku nie ma tam, gdzie powinien być. Serwer został wyłączony. Niezależnie od tego, jak dobrym jesteś programistą, nie jesteś w stanie kontrolować *wszystkiego*. Kiedy tworzysz metodę, której działanie jest opatrzone ryzykiem niepowodzenia, musisz także stworzyć kod, który obsłuży potencjalnie niebezpieczną sytuację. Ale skąd wiadomo, że metoda może nieść ze sobą potencjalne niebezpieczeństwo? Gdzie umieszczać kod *obsługujący* **wyjątkowe** sytuacje? W *tym* rozdziale stworzymy odtwarzacz MIDI wykorzystujący ryzykowny interfejs programistyczny `JavaSound`, zatem lepiej dowiedzmy się, jak zabezpieczyć się przed potencjalnymi niebezpieczeństwami.

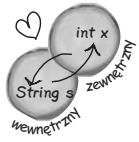


12 Historia bardzo graficzna

Musisz się z tym pogodzić — tworzenie interfejsów graficznych jest konieczne. Nawet jeśli wierzysz, że już do końca życia będziesz pisać programy działające na serwerze, to jednak będziesz także musiał pisać programy narzędziowe i zapewne będziesz chciał, aby miały one jakiś interfejs graficzny. Zagadnieniom interfejsu graficznego poświęcimy dwa rozdziały, w których przedstawimy także inne cechy języka, takie jak obsługa zdarzeń oraz klasy wewnętrzne. Naciśniemy przycisk na ekranie, wskażemy coś myszą, wyświetlimy obraz zapisany w formacie JPEG, a nawet stworzymy małą animację.

```
class MojaKlasaZewnetrzna {

  class MojaKlasaWewnetrzna {
    void doDziela() {
    }
  }
}
```

Klasa wewnętrzna jest całkowicie zawarta w klasie zewnętrznej.



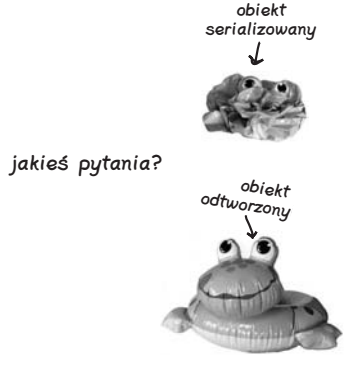
Te dwa obiekty przebywające na stercie łączy szczególna więź. Obiekt wewnętrzny może korzystać ze składowych obiektu zewnętrznego (i na odwrót).

13 Popracuj nad Swingiem

Swing jest łatwy. Chyba że naprawdę zwracasz uwagę na to, co jest gdzie wyświetlane. Kod wykorzystujący Swing *wydaje* się prosty, ale kiedy go skompilujesz, uruchomisz i popatrzysz na wyniki, to często będziesz mógł sobie pomyśleć: „Hej, przecież to nie miało być wyświetlone w *tym* miejscu". To, co zapewnia prostotę kodu, sprawia jednocześnie, że trudne jest kontrolowanie położenia elementów — tym „czymś" jest **menedżer układu**. Jednak przy odrobinie pracy można nagiąć menedżer układu do naszej woli. W tym rozdziale popracujemy nad naszym Swingiem i dowiemy się czegoś więcej o różnych elementach graficznych.

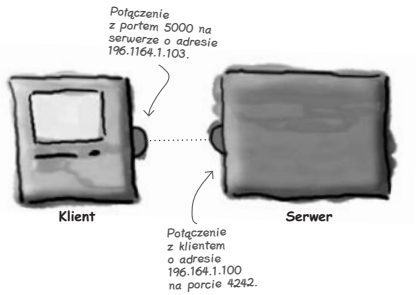
14 Zapisywanie obiektów

Obiekty można pakować i odtwarzać. Obiekty mają swój stan i działanie. Działanie obiektu określa jego klasa, natomiast *stan* zależy od konkretnego *obiektu*. Jeśli program musi zapisać stan obiektu, *możesz to zrobić w sposób trudny* — analizując każdy obiekt i pracowicie zapisując wartości wszystkich właściwości. **Możesz także zapisać obiekt w sposób łatwy i obiektowy** — „zamrażając" obiekt (przeprowadzając jego serializację), a następnie odtwarzając (poprzez jego deserializację).



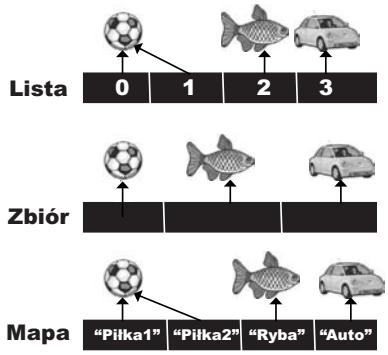
15 Nawiąż połączenie

Nawiąż połączenie ze światem zewnętrznym. To takie łatwe. Wszelkie szczegóły związane z komunikacją sieciową na niskim poziomie są obsługiwane przez klasy należące do biblioteki java.net. Jedną z najlepszych cech Javy jest to, iż wysyłanie i odbieranie danych przez sieć to w zasadzie normalne operacje wejścia-wyjścia, z tą różnicą, że są w nich wykorzystywane nieco inne strumienie. W tym rozdziale stworzymy gniazda używane przez programy klienckie. Stworzymy także gniazda używane przez programy pełniące funkcje serwerów. Napiszemy zarówno program klienta, jak i serwera. Zanim skończysz czytać ten rozdział, będziesz już dysponować w pełni funkcjonalnym, wielowątkowym programem do prowadzenia internetowych pogawędek. Zaraz... czy właśnie padło słowo *wielowątkowy*?



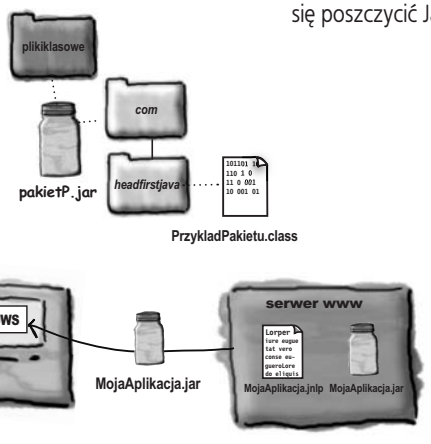
16 Struktury danych

Sortowanie w Javie to pestka. Dysponujesz wszystkimi narzędziami do gromadzenia i manipulowania danymi i to bez konieczności pisania własnych algorytmów. Biblioteka kolekcji Javy (*Java Collections Framework*) posiada struktury danych, które powinny sprostać praktycznie wszystkiemu, co będziesz chciał zrobić. Potrzebujesz listy, do której będziesz mógł łatwo dodawać elementy? Chcesz znaleźć coś na podstawie nazwy? A może chcesz stworzyć listę, która automatycznie będzie usuwać powtarzające się elementy? Albo posortować listę pracowników na podstawie liczby „przyjacielskich klepnięć w plecy"?



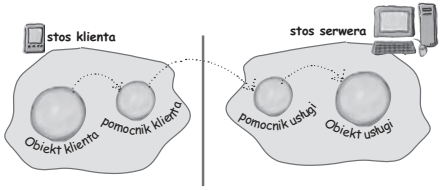
17 Rozpowszechnij swój kod

Już czas wypłynąć na „szerokie wody". Napisałeś kod swojej aplikacji. Przetestowałeś go. Udoskonaliłeś. Powiedziałeś wszystkim znajomym, że świetnie by było, gdybyś już w życiu nie musiał napisać choćby jednej linijki kodu. Ale w końcu stworzyłeś dzieło sztuki. Przecież Twoja aplikacja działa! W ostatnich dwóch rozdziałach książki zajmiemy się zagadnieniami związanymi z organizowaniem, pakowaniem i wdrażaniem kodu. Przyjrzymy się możliwościom wdrażania lokalnego, mieszanego i zdalnego, w tym wykonywalnym plikom Jar, technologii Java Web Start, RMI oraz serwletom. Nie stresuj się! Niektóre z najbardziej niesamowitych rozwiązań, jakimi może się poszczycić Java, są prostsze, niż można by przypuszczać.



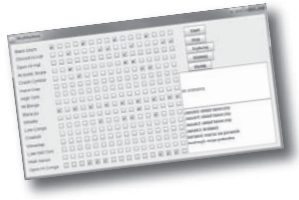
18 Przetwarzanie rozproszone

Zdalne wykonywanie aplikacji nie zawsze jest złe. Oczywiście to prawda, że życie *jest* łatwiejsze, gdy wszystkie elementy aplikacji znajdują się w jednym miejscu i są zarządzane przez jedną wirtualną maszynę Javy. Jednak takie rozwiązanie nie zawsze jest możliwe. Co w sytuacji, gdy aplikacja realizuje bardzo złożone obliczenia? Co, jeśli potrzebuje informacji pochodzących z zabezpieczonej bazy danych? W tym rozdziale przedstawiona zostanie zadziwiająco prosta technologia wywoływania zdalnych metod — *RMI* (ang. *Remote Method Invocation*). Pobieżnie przyjrzymy się także innym rozwiązaniom — serwletom, komponentom *EJB* (ang. *Enterprise Java Bean*) oraz technologii Jini.



A Dodatek A

Ostatnie doprawianie kodu. Kompletny kod aplikacji `MuzMachina` w ostatecznej wersji klient-serwer z możliwością prowadzenia muzycznych pogawędek. Twoja szansa, by zostać gwiazdą rocka.



B Dodatek B

Dziesięć najważniejszych zagadnień, które niemal znalazły się w tej książce… Jeszcze nie możemy Cię zostawić i wypuścić w świat. Mamy dla Ciebie jeszcze kilka dodatkowych informacji, jednak to już *jest* koniec tej książki. I tym razem mówimy to zupełnie serio.

S Skorowidz

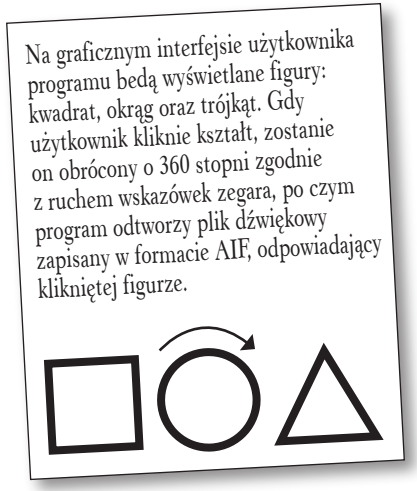
Wycieczka do Obiektowa

Mówiono mi, że będą obiekty. W rozdziale 1. cały tworzony kod był umieszczany w metodzie `main()`. Nie jest to poprawne rozwiązanie obiektowe. W rzeczywistości, z punktu widzenia programowania zorientowanego obiektowo, jest to rozwiązanie *całkowicie* niewłaściwe. Cóż, w programie „krasomówczym" wykorzystaliśmy *kilka* obiektów, takich jak tablice łańcuchów znaków (obiektów `String`), jednak nie stworzyliśmy żadnego własnego *typu obiektowego*. Zatem teraz musimy zostawić świat programowania proceduralnego, usunąć to co najważniejsze z metody `main()` i zacząć tworzyć własne obiekty. Przyjrzymy się czynnikom, które sprawiają, że programowanie zorientowane obiektowo przy użyciu języka Java, jest takie fajne. Przedstawimy różnice pomiędzy *klasą* a *obiektem*. Przekonamy się, w jaki sposób obiekty mogą ułatwić nam życie (a przynajmniej jego aspekty związane z programowaniem; nie będziemy bowiem w stanie sprawić, abyś zaczął się znać na modzie i wyrobił sobie dobry gust). Jedno ostrzeżenie: kiedy już dotrzesz do Obiektowa, możesz już nigdy się z niego nie wydostać. Wyślij nam pocztówkę.

Wojna o fotel
(albo Jak Obiekty Mogą Zmienić Twoje Życie)

Dawno temu w sklepie z oprogramowaniem dwóch programistów dostało tę samą specyfikację i kazano im „napisać co trzeba”. Naprawdę Denerwujący Szef Projektu zmusił obu programistów do rywalizowania z sobą, obiecując im, że ten, który pierwszy odda kod, dostanie Superfotel™, który mają wszyscy programiści w Dolinie Krzemowej. Bronek — programista proceduralny — oraz Jurek — programista obiektowy — wiedzą, że wykonanie zadania będzie jak przysłowiowa „kaszka z mleczkiem”.

specyfikacja



Bronek, siedząc w swoim boksie, pomyślał: „Jakie rzeczy ten program ma *robić*? Jakich ***procedur*** potrzebuje?”. Po czym sam sobie odpowiedział: „Potrzebne mi będą procedury: **obróć** i **odtwórzDźwiek**”. I zabrał się za pisanie odpowiednich procedur. No bo w końcu czym innym *jest* program, jeśli nie zbiorem stosownych procedur?

W międzyczasie Jurek poszedł do kawiarni na filiżankę kawy i zadał sobie pytanie: „Jakie są ***obiekty*** w tym programie... jacy są jego najważniejsi *bohaterowie*?”. Pierwszą odpowiedzią, jaka mu przyszła do głowy, była: **Figury**. Oczywiście w programie występują też inne obiekty, takie jak Użytkownik, Dźwięk czy też zdarzenie Kliknięcie, ale Jurek dysponuje już biblioteką obsługującą te obiekty, zatem może się skoncentrować na tworzeniu Figur. Przeczytaj dalszą treść rozdziału, aby się przekonać, jak Jurek i Bronek tworzyli swoje programy, a przede wszystkim, aby uzyskać odpowiedź na najbardziej palące pytanie: ***„Kto dostanie Superfotel?”***.

fotel

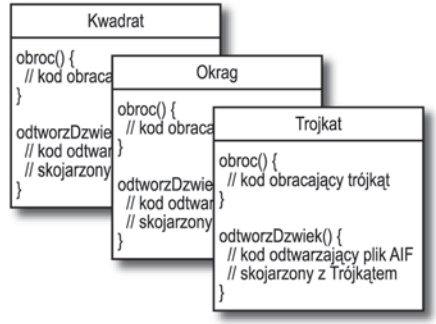
W boksie Bronka

Jak miliardy razy wcześniej Bronek zabrał się do pisania **Niezwykle Ważnych Procedur**. Błyskawicznie napisał procedury **obroc** oraz **odtworzDzwiek**:

```
obroc(numFigury) {
  // obrócenie figury o 360 stopni
}
odtworzDzwiek(numFigury) {
  // na podstawie numeru figury określ,
  // jaki plik AIF należy odtworzyć i odtwórz go
}
```

W kawiarni na laptopie Jurka

Jurek stworzył ***klasy*** dla każdej z trzech figur.



Bronek myślał, że już ma w garści wygraną. Już niemal mógł czuć miękką skórę Superfotela pod swoim...

Ale chwileczkę! Nastąpiła zmiana specyfikacji.

— W porządku. *Technicznie* rzecz biorąc, wygrał Bronek — powiedział Szef — ale musimy dodać do programu jedną malutką rzecz. Dla takich doświadczonych programistów jak wy, nie będzie to stanowiło żadnego problemu.

— *Gdyby dostawać dziesięć groszy za każdą zmianę specyfikacji...* — pomyślał Bronek.
— *A jednak Jurek jest dziwnie spokojny. Co jest grane?* Jednak Bronek wciąż trzymał się swojej opinii, że podejście obiektowe, choć ciekawe, jest wolne. Pomyślał też, że jeśli ktoś chciałby sprawić, by zmienił zdanie, musiałby poddać go bezlitosnym torturom.

Oto, co zostało dodane do specyfikacji

Z powrotem w boksie Bronka

Procedura obracania nawet po zmianach będzie działać dobrze, podobnie jak kod służący do przejrzenia tablicy figur i dopasowania wartości numFigury do faktycznej figury. ***Ale trzeba zmienić procedurę odtworzDzwiek.*** I co to jest ten cały plik *.hif*?

```
odtworzDzwiek(numFigury) {
  // jeśli kształt to nie "ameba",
     // to na podstawie numeru figury określ,
     // jaki plik AIF należy odtworzyć i odtwórz
     // go
  // w przeciwnym razie
     // odtwórz plik dźwiękowy .hif skojarzony
     // z "amebą"
 }
```

Okazało się, że zmiany nie są takie straszne, ***niemniej jednak konieczność modyfikowania już sprawdzonego kodu spowodowała u Bronka pewne uczucie niepokoju***. W końcu właśnie on, jak nikt inny, powinien doskonale wiedzieć, że niezależnie od tego co mówi szef projektu, ***specyfikacja zawsze się zmienia***.

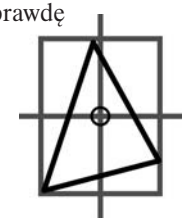
Na plaży na laptopie Jurka

Jurek, uśmiechnął się, wychylił łyczek swojej Margherity i *napisał jedną nową klasę*. Czasami sobie myślał, że rzeczą, którą najbardziej kocha w programowaniu obiektowym, jest brak konieczności modyfikowania kodu, który już raz został przetestowany i udostępniony.

Elastyczność, rozszerzalność… — mruczał pod nosem Jurek, przypominając sobie wszystkie zalety programowania obiektowego.

Bronek wpadł do biura Szefa tuż przed Jurkiem

(Acha! I tyle są warte te wszystkie obiektowe bzdury). Ale uśmiech na jego twarzy szybko zgasł, kiedy Naprawdę Denerwujący Szef Projektu powiedział: „Ależ nie! Ameba miała być obracana w zupełnie *inny* sposób…”.



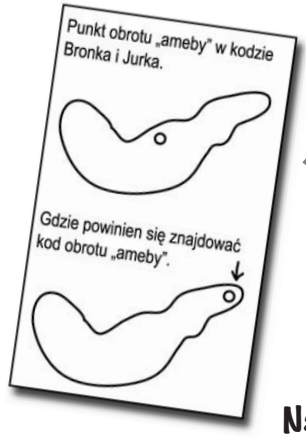
Okazuje się, że kod obracający figury napisany przez obu programistów działa w następujący sposób:

1) Określa prostokątny obszar, w jakim mieści się figura.

2) Wyznacza środek tego obszaru i obraca go wokół tego punktu.

Ale, jak się okazuje, kształt przypominający amebę miał być obracany wokół punktu znajdującego się na jego *końcu*, zupełnie tak samo jak wskazówka zegara.

— Jestem ugotowany — pomyślał Bronek, wyobrażając sobie bulgoczący KociołekSzamana™. — Chociaż, niech pomyślę… Mógłbym przecież dodać do procedury obracającej figury jeszcze jedną instrukcję `if-else` i w niej zakodować na stałe punkt obrotu dla ameby. To prawdopodobnie nie zepsułoby całej procedury. Ale wtedy cichutki głosik w jego głowie odezwał się: — *Wielki błąd. Czy jesteś absolutnie pewny, że specyfikacja znowu się nie zmieni?*



O tym beztrosko zapomniano napisać w specyfikacji

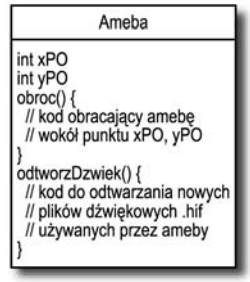
Z powrotem w boksie Bronka

Bronek doszedł do wniosku, że lepszym rozwiązaniem będzie dodanie do procedury obracającej figury argumentów, które określają współrzędne punktu obrotu. ***Wprowadziło to poważne zmiany w kodzie.*** Testowanie, rekompilacja, cała masa roboty, którą trzeba wykonać od nowa. To, co wcześniej działało, teraz przestało działać.

```
  obroc(numFigury, xPt, yPt) {
  // jeśli figura to nie "ameba",
     // wyznaczenie środka na podstawie
        prostokąta opisanego
     // i obrócenie figury o 360 stopni
  // w przeciwnym razie
     // wyznaczenie punktu obrotu z uwzględnieniem
     // podanego przesunięcia xPt, yPt i
     // obrócenie figury o 360 stopni
}
```

Na laptopie Jurka, gdzieś na widowni Festiwalu Kapel Wirtualnych

Nie tracąc nawet jednego taktu z prezentowanych utworów, Jurek zmodyfikował **metodę** obracającą, ale wyłącznie w klasie Ameba. ***W żaden sposób nie zmienił żadnego z przetestowanych, skompilowanych i działających kodów*** stanowiących pozostałe części programu. Aby określić punkt obrotu figury przypominającej amebę, Jurek dodał **atrybuty**, które będą mieć wszystkie „ameby”. Zmodyfikował, przetestował i przesłał (oczywiście bezprzewodowo) zmodyfikowaną wersję programu w czasie trwania utworu *Serce metody*.



Czyli to Jurek — „obiektowiec" — zdobył Superfotel, czy tak?

Nie tak szybko. Bronek znalazł pewną wadę rozwiązania przedstawionego przez Jurka. A ponieważ uważał, że jeśli zdobędzie Superfotel, to zdobędzie także panią Lucynkę z księgowości, musiał zatem przystąpić do natarcia.

BRONEK: W twoim programie powtarzają się te same fragmenty kodu! Procedura do obracania jest we wszystkich tych rzeczach… no — figurach.

JUREK: To jest ***metoda***, a nie *procedura*. A to nie są rzeczy tylko ***klasy***.

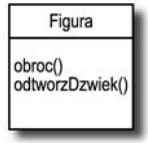
BRONEK: Nieważne. Idiotyczny projekt. Musisz mieć aż *cztery* różne „metody" do obracania. Czy taki projekt może być dobry?

JUREK: O! Jak mniemam, nie widziałeś ostatecznej wersji. Pozwól, że ci pokaże coś, co w programowaniu obiektowym określamy jako **dziedziczenie**.

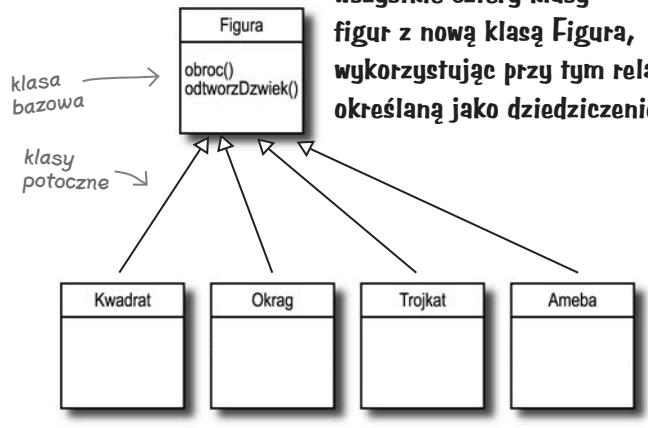
To o niej marzył Bronek (zakładał, że Superfotel zrobi odpowiednie wrażenie)

1 Wybrałem elementy, które są wspólne dla wszystkich czterech klas.

2 To wszystko są Figury, wszystkie je można obracać, a ich kliknięcie powoduje obrót i odtworzenie pliku dźwiękowego. Dlatego też wyodrębniłem ich wspólne cechy i umieściłem je w nowej klasie o nazwie Figura.



3 Następnie połączyłem wszystkie cztery klasy figur z nową klasą Figura, wykorzystując przy tym relację określaną jako dziedziczenie.



Należy to rozumieć jako: **„Kwadrat dziedziczy po Figurze"**, **„Okrąg dziedziczy po Figurze"** i tak dalej. Usunąłem metody `obroc()` i `odtworzDzwiek()` z pozostałych klas, dzięki czemu teraz jest tylko jedna kopia tej metody.

Klasa `Figura` jest nazywana klasą **bazową** dla pozostałych czterech klas. Natomiast te cztery klasy, są klasami **potomnymi** klasy `Figura`. Klasy potomne dziedziczą metody klasy bazowej. Innymi słowy, *jeśli klasa Figura ma jakieś możliwości funkcjonalne, to możliwości te są automatycznie dostępne w klasach potomnych.*

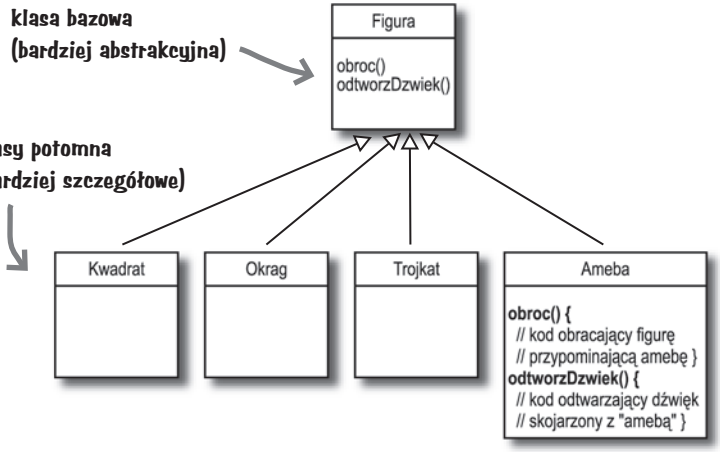
A co z metodą obroc() dla „ameby"?

BRONEK: Ale na czym polegał cały problem — czy nie na tym, że figura przypominająca kształtem amebę miała mieć całkowicie inne procedury `obroc()` i `odtworzDzwiek()`?

JUREK: Metody.

BRONEK: Nieważne. W jaki sposób „ameba" może robić coś innego, jeśli dziedziczy możliwości funkcjonalne po klasie `Figura`?

JUREK: To ostatni etap zadania. Klasa `Ameba` może przesłonić metody klasy `Figura`. Następnie, podczas wykonywania programu, kiedy każemy obrócić „amebę", JVM będzie dokładnie wiedzieć, jaką metodę `obroc()` należy wykonać.



4

W klasie Ameba przesłaniam metody obroc() i odtworzDzwiek() bazowej klasy Figura.

Przesłanianie oznacza po prostu, że jeśli klasa potomna musi zmienić lub rozszerzyć działanie jednej z odziedziczonych metod, to po prostu ją ponownie definiuje.

BRONEK: A w jaki sposób „każesz" obiektowi Ameba coś zrobić? Czy nie musisz wywołać jakiejś funkcji, o przepraszam — *metody* — i wskazać jej, *którą* figurę obrócić?

JUREK: I to właśnie jest najfajniejsza rzecz w programowaniu obiektowym. Kiedy trzeba obrócić, na przykład, trójkąt, kod programu wywołuje metodę `obroc()` *obiektu trójkąta*. Pozostała część programu tak naprawdę wcale nie interesuje się tym, w jaki sposób obraca się trójkąt. A kiedy trzeba dodać coś nowego do programu, tworzy się nową klasę dla nowego typu obiektu; dzięki temu ten **nowy obiekt będzie się zachowywać w sposób unikalny**.

Ta niepewność mnie zabije! Kto wygra Superfotel?

Ania z drugiego piętra.

(bez wiedzy kogokolwiek innego Szef Projektu przekazał specyfikację programu *trzem* programistom)

Co Ci się podoba w programowaniu obiektowym?

„Pomaga mi projektować programy w bardziej naturalny sposób. Obiekty wchodzące w skład programów mogą ewoluować."

— Jonasz, 27, projektant oprogramowania

„To, że w razie konieczności dodania nowych możliwości nie muszę ingerować w kod, który już został napisany i przetestowany."

— Bartek, 32, programista

„Podoba mi się, że dane i metody, które na tych danych operują, są zgrupowane w jednej klasie."

— Jonatan, 22, miłośnik piwa

„Podoba mi się możliwość wykorzystywania kodu w innych aplikacjach. Tworząc nową klasę, mogę ją napisać w sposób na tyle elastyczny, że będzie ją można wykorzystać w przyszłości w innych programach."

— Krzysiek, 39, menedżer projektu

„Nie mogę uwierzyć, że Krzysiek coś takiego powiedział. Nie napisał nawet jednej linijki kodu od pięciu lat."

— Darek, 44, pracuje dla Krzyśka

„Oprócz możliwości zdobycia Superfotela?"

— Ania, 34, programistka

WYSIL SZARE KOMÓRKI

Nadszedł czas, żeby trochę rozruszać neurony.

Przeczytałeś właśnie opowieść o programiście proceduralnym konkurującym z programistą obiektowym. Przy okazji mogłeś się przyjrzeć krótkiej prezentacji kluczowych pojęć związanych z programowaniem obiektowym, takich jak klasy, metody oraz atrybuty. Dalsza część rozdziału zostanie poświęcona dokładniejszej prezentacji klas i obiektów (do zagadnień dziedziczenia i przesłaniania metod powrócimy w kolejnych rozdziałach).

Bazując na informacjach zdobytych do tej pory (a może także na wiedzy zdobytej podczas korzystania z innego języka zorientowanego obiektowo), poświęć chwilkę na przemyślenie i próbę podania odpowiedzi na poniższe pytania:

Jakie są podstawowe zagadnienia, które należy przemyśleć, projektując klasy w języku Java? Jakie pytania należy sobie przy tym zadać? Gdybyś miał stworzyć listę rzeczy, które należy zrobić podczas projektowania nowych klas, to co znalazłoby się na tej liście?

Metapoznaniowa podpowiedź

Jeśli utknąłeś i nie możesz znaleźć rozwiązania ćwiczenia, spróbuj porozmawiać o nim z sobą na głos. Mówienie (i słuchanie) aktywizuje różne części mózgu. Choć metoda ta daje najlepsze rezultaty, kiedy można porozmawiać z inną osobą, to jednak można także porozmawiać z ulubionym zwierzakiem. To właśnie w ten sposób nasz pies dowiedział się, co to jest polimorfizm.

Projektując klasę, myśl o obiektach, które będą tworzone na podstawie tego typu. Pomyśl o:

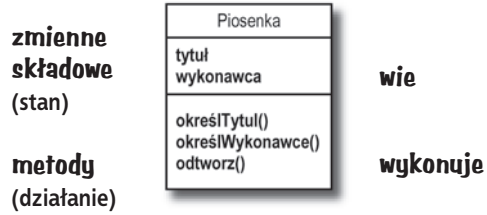
- informacjach, jakie obiekt **zna**,
- czynnościach, jakie obiekt **wykonuje**.

Informacje, jakie obiekt ma o sobie, są nazywane

- składowymi

Czynności, jakie obiekt jest w stanie wykonywać, są nazywane

- metodami



Informacje, jakie obiekt ***wie***, na swój temat, nazywamy **składowymi**. Reprezentują one stan obiektu (dane) i mogą przyjmować unikalne wartości w każdym z obiektów danego typu.

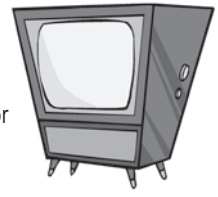
Pamiętaj, że mówiąc o kopii, mamy na myśli obiekt.

Czynności, jakie obiekt *może wykonywać*, nazywamy **metodami**. Projektując klasę, należy określić, jakie informacje na swój temat obiekt musi znać, jak również zaprojektować metody, które będą operować na tych danych. Bardzo często się zdarza, że obiekty mają metody służące do ustawiania oraz odczytywania wartości składowych. Na przykład obiekt Alarm ma składową określającą czas alarmu oraz dwie metody służące do określania i pobierania tego czasu.

A zatem obiekty mają kopie, składowe oraz metody, jednak zarówno składowe, jak i metody stanowią część klasy.

Zaostrz ołówek

Poniżej wpisz, co obiekt Telewizor powinien wiedzieć i robić.

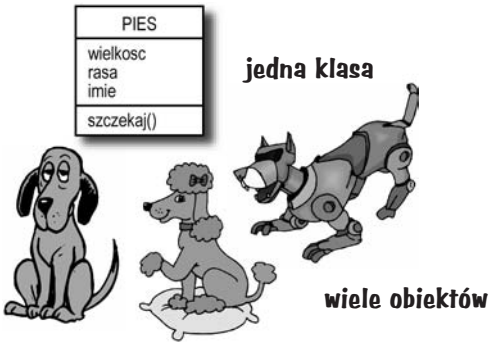


Telewizor

zmienne składowe

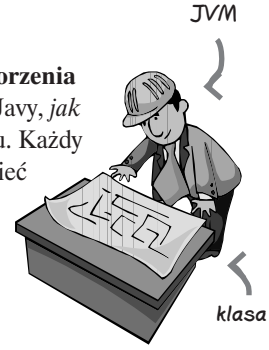
metody

Jaka jest różnica pomiędzy klasą a obiektem?



Klasa nie jest obiektem (jednak służy do ich tworzenia)

Klasa jest jak gdyby *matrycą* służącą do tworzenia obiektów. Informuje ona wirtualną maszynę Javy, *jak* należy utworzyć obiekt tego konkretnego typu. Każdy obiekt utworzony na podstawie klasy może mieć unikalne wartości składowych. Na przykład, można użyć klasy `Przycisk` do stworzenia kilkunastu różnych przycisków, z których każdy będzie mieć inny kolor, wielkość, kształt, etykietę i tak dalej.



Wyobraź to sobie w następujący sposób...

Obiekt można by porównać z jednym wpisem w książce adresowej.

Jedną z możliwych analogii obiektów są puste karteczki, które można wpisać do kołowego notatnika z adresami. Każda z takich karteczek ma takie same pola do wypełnienia (odpowiadające składowym obiektu). Wypełniając ją, tworzymy kopię (obiekt), a podane na niej informacje określają stan kopii.

Metody klasy to czynności, jakie można wykonywać na konkretnej karteczce; klasa `NotatnikAdresowy` mogłaby mieć następujące metody: `pobierzNazwisko()`, `zmienNazwisko()`, `okreslNazwisko()`.

A zatem, każda karteczka może wykonywać te same operacje (pobrać zapisane na niej nazwisko, zmienić je i tak dalej), jednak każda z nich zawiera unikalne *informacje*, dostępne wyłącznie na niej.

Tworzenie pierwszego obiektu

A zatem, czego będziesz potrzebował do stworzenia i wykorzystania swojego pierwszego obiektu? Będą Ci potrzebne *dwie* klasy. Pierwsza będzie klasą obiektu, którego chcesz użyć (Pies, Budzik, Telewizor i tak dalej), natomiast druga posłuży do *przetestowania* nowej klasy. To właśnie w tej klasie *testującej* zostanie umieszczona metoda `main()`, a w niej będzie tworzony obiekt Twojej nowej klasy. Klasa testująca ma tylko jedno zadanie — *przetestować* metody i składowe obiektu Twojej nowej klasy.

Zaczynając od tego miejsca, w wielu przykładach przedstawionych w dalszej części książki będziesz mógł znaleźć dwie klasy. Pierwsza z nich będzie tą *właściwą* — czyli klasą, której obiektów chcemy używać; z kolei druga będzie klasą *testującą*, a jej nazwa będzie odpowiadać nazwie klasy właściwej z dodanym na końcu słowem **`Tester`**. Na przykład, jeśli stworzymy klasę **`Bungee`**, to będzie nam potrzebna także klasa **`BungeeTester`**. Wyłącznie klasa *<jakaśTamKlasa>***`Tester`** będzie posiadać metodę `main()`, a jedynym celem jej istnienia będzie stworzenie obiektów nowej klasy (nie klasy testującej) i wykorzystanie operatora kropki (`.`) w celu uzyskania dostępu do metod i składowych tych obiektów. Wszystkie te zasady staną się całkowicie jasne, gdy przeanalizujesz poniższe przykłady.

Operator kropki (.)

Operator kropki (.) zapewnia dostęp do stanu oraz do zachowania obiektu (składowych i metod).

```
// tworzymy nowy obiekt
Pies p = new Pies();

// każemy psu szczekać,
// dodając do zmiennej p
// operator kropki w celu
// wywołania metody
// szczekaj()
p.szczekaj();

// określamy wielkość psa,
// także tym razem
// używając operatora
// kropki (.)
p.wielkosc = 40;
```

1 Napisz kod klasy.

```
class Pies {
  int wielkosc;
  String rasa;
  String imie;

  void szczekaj() {
    System.out.println("Chau! Chauuu!");
  }
}
```

składowe

metoda

2 Napisz klasę testującą (Tester).

```
class PiesTester {
  public static void main (String[] args) {
    // Tutaj umieścimy kod
    // testujący klasę Pies
  }
}
```

klasa posiada tylko metodę main() (w następnym kroku umieścimy w tej metodzie jakiś kod)

3 W klasie testującej stwórz obiekt i użyj jego składowych i metod.

```
class PiesTester {
  public static void main (String[] args) {
    Pies p = new Pies();
    p.wielkosc = 40;
    p.szczekaj();
  }
}
```

utwórz obiekt Pies

operator kropki

użyj operatora kropki (.) w celu określenia wielkości Psa

i wywołania jego metody szczekaj()

Jeśli już masz jakieś rozeznanie w programowaniu obiektowym, to będziesz wiedział, że nie używamy hermetyzacji. Tym zagadnieniem zajmiemy się w rozdziale 4.

Tworzenie i testowanie obiektów Film


```
class Film {
  String tytul;
  String rodzaj;
  int ocena;

  void odtworz() {
    System.out.println("Odtwarzamy film.");
  }
}

public class FilmTester {
  public static void main(String[] args) {
    Film pierwszy = new Film();
    pierwszy.tytul = "Przeminęło z hossą";
    pierwszy.rodzaj = "Tragedia";
    pierwszy.ocena = -2;
    Film drugi = new Film();
    drugi.tytul = "Matrix dla zuchwałych";
    drugi.rodzaj = "Komedia";
    drugi.ocena = 5;
    drugi.odtworz();
    Film trzeci = new Film();
    trzeci.tytul = "Byte Club";
    trzeci.rodzaj = "Tragedia, ale o wydźwięku optymistycznym";
    trzeci.ocena = 127;
  }
}
```

Zaostrz ołówek

Klasa `FilmTester` tworzy trzy obiekty (kopie) klasy `Film`, a następnie, przy wykorzystaniu operatora kropki (`.`), przypisuje konkretne wartości ich składowym. Klasa ta wywołuje także metodę jednego z tych obiektów. Na rysunku z prawej strony w pustych miejscach wpisz wartości, jakie będą miały odpowiednie składowe obiektów pod koniec działania metody `main()`.

obiekt 1
- tytuł
- rodzaj
- ocena

obiekt 2
- tytuł
- rodzaj
- ocena

obiekt 3
- tytuł
- rodzaj
- ocena

Szybko! Opuszczamy metodę main!

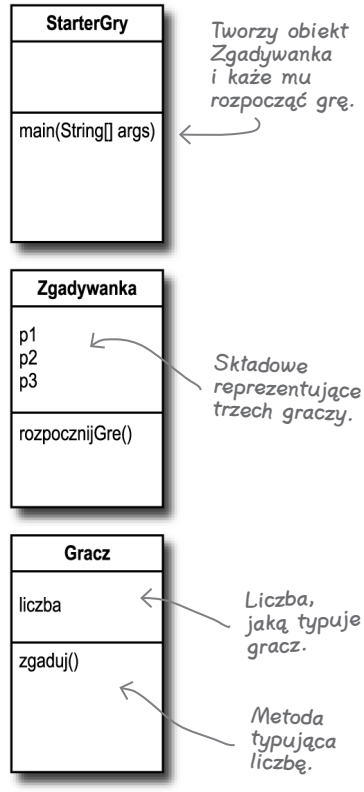
Dopóki działasz w obrębie metody `main()`, dopóty tak naprawdę nie dotarłeś do Obiektowa. Wykonywanie operacji w tej metodzie jest dobre dla prostego programu testowego, jednak prawdziwe aplikacje pisane obiektowo wymagają, aby obiekty komunikowały się z innymi obiektami, a nie by były tworzone i testowane w jakiejś statycznej metodzie.

Dwa zastosowania metody main:

- do **testowania** klas wykorzystywanych w aplikacji,
- do **uruchamiania** bądź wykonywania **aplikacji**.

Prawdziwa aplikacja napisana w Javie to w zasadzie nic innego jak grupa obiektów, które komunikują się pomiędzy sobą. W tym przypadku *komunikowanie się* oznacza wywoływanie metod obiektów. Na poprzedniej stronie (jak również w 4. rozdziale książki) metoda `main()` została umieszczona w niezależnej klasie `Tester` i służyła do utworzenia i sprawdzenia metod i zmiennych innych klas. W rozdziale 6. przyjrzymy się wykorzystaniu klasy, w której metoda `main()` służy do uruchomienia *prawdziwej* aplikacji napisanej w Javie (czyli, między innymi, do utworzenia obiektów i zapewnienia im możliwości interakcji z innymi obiektami).

Jak na razie przedstawimy jednak prosty przykład tego, jak może działać prawdziwa aplikacja Javy. Ponieważ wciąż znajdujemy się na samym początku nauki Javy, nasz warsztat jest bardzo ograniczony, dlatego też uznasz zapewne, że przedstawiony program jest nieco głupkowaty i nieefektywny. Możesz się zastanowić, co zrobić, aby go poprawić; swoją drogą, dokładnie to zrobimy w następnych rozdziałach. Nie przejmuj się, jeśli będziesz mieć trudności ze zrozumieniem fragmentów kodu — jego podstawowym celem jest przedstawienie komunikacji pomiędzy obiektami.



Zgadywanka

Podsumowanie:

Nasza gra — zgadywanka — wykorzystuje obiekt „gry" oraz trzy obiekty „graczy". Gra generuje liczbę losową z zakresu od 0 do 9, a trzej gracze starają się ją odgadnąć. (Nikt nie mówił, że to ma być *pasjonująca* gra).

Klasy:

```
Zgadywanka.class Gracz.class StarterGry.class
```

Logika działania:

Działanie aplikacji rozpoczyna się w klasie `StarterGry`; klasa ta posiada metodę `main()`.

W metodzie `main()` jest tworzony obiekt `Zgadywanka`, a następnie zostaje wywołana metoda `rozpocznijGre()`.

Cała gra odbywa się wewnątrz metody `rozpocznijGre()` obiektu `Zgadywanka`. Metoda ta tworzy trzech graczy, po czym „wymyśla" losową liczbę (którą gracze mają odgadnąć). Następnie metoda prosi graczy o odgadnięcie liczby, sprawdza podane przez nich wartości i wyświetla informacje o zwycięzcy (zwycięzcach) albo prosi o ponowną próbę odgadnięcia.

```
class Zgadywanka {
  Gracz p1;
  Gracz p2;
  Gracz p3;

  public void rozpocznijGre() {
    p1 = new Gracz();
    p2 = new Gracz();
    p3 = new Gracz();

    int typp1 = 0;
    int typp2 = 0;
    int typp3 = 0;

    boolean p1odgadl = false;
    boolean p2odgadl = false;
    boolean p3odgadl = false;

    int liczbaOdgadywana = (int) (Math.random() * 10);
    System.out.println("Myślę o liczbie z zakresu od 0 do 9...");

    while(true) {
      System.out.println("Należy wytypować liczbę: " + liczbaOdgadywana );

      p1.zgaduj();
      p2.zgaduj();
      p3.zgaduj();

      typp1 = p1.liczba;
      System.out.println("Gracz pierwszy wytypował liczbę: " + typp1);

      typp2 = p2.liczba;
      System.out.println("Gracz drugi wytypował liczbę: " + typp2);

      typp3 = p3.liczba;
      System.out.println("Gracz trzeci wytypował liczbę: " + typp3);

      if (typp1 == liczbaOdgadywana) {
        p1odgadl = true;
      }
      if (typp2 == liczbaOdgadywana) {
        p2odgadl = true;
      }
      if (typp3 == liczbaOdgadywana) {
        p3odgadl = true;
      }

      if (p1odgadl || p2odgadl || p3odgadl) {

        System.out.println("Mamy zwycięzcę!");
        System.out.println("Czy gracz pierwszy wytypował poprawnie? " + p1odgadl);
        System.out.println("Czy gracz drugi wytypował poprawnie? " + p2odgadl);
        System.out.println("Czy gracz trzeci wytypował poprawnie? " + p3odgadl);
        System.out.println("Koniec gry.");
        break; // Gra skończona, zatem wychodzimy z pętli while

      } else {
        System.out.println("Gracze będą musieli spróbować jeszcze raz.");
      } // koniec if - else
    } // koniec while
  } // koniec metody rozpocznijGre
} // koniec klasy
```

Klasa Zgadywanka ma trzy składowe służące do przechowywania trzech obiektów Gracz.

Utworzenie trzech obiektów Gracz i zapisanie ich w trzech składowych.

Deklaracja trzech zmiennych, w których będą przechowywane trzy liczby wytypowane przez poszczególnych graczy.

Deklaracja trzech zmiennych, które będą przechowywać wartości true (prawda) lub false (fałsz), w zależności od odpowiedzi konkretnego gracza.

Wyznaczenie liczby, jaką będą musieli odgadnąć gracze.

Wywołanie metody zgaduj() każdego z graczy.

Pobranie liczb wytypowanych przez każdego z graczy (wyników wywołania metody zgaduj()) poprzez odczytanie jej ze składowych obiektów graczy.

Sprawdzenie liczb wytypowanych przez graczy w celu określenia, czy odpowiadają one wyznaczonej liczbie. Jeśli gracz wytypował poprawnie, to odpowiedniej zmiennej przypisywana jest wartość true (pamiętaj, że domyślnie zmienna ta ma wartość false).

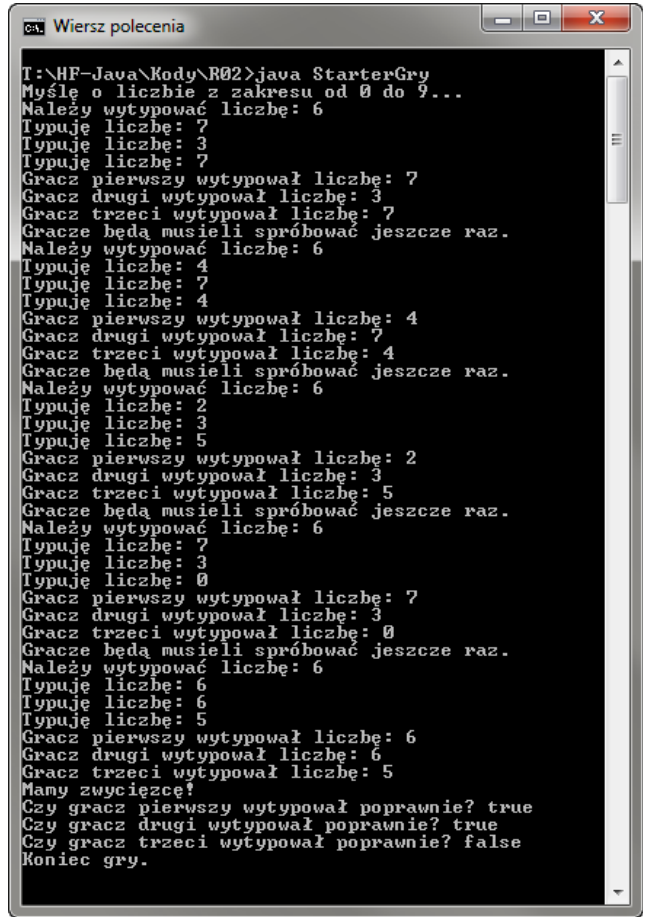
Jeśli gracz pierwszy LUB gracz drugi, LUB gracz trzeci odgadł... (operator || to logiczne LUB).

W przeciwnym przypadku pętla jest dalej realizowana, a gracze są proszeni o wytypowanie kolejnych liczb.

Uruchamianie zgadywanki

```
class Gracz {
  int liczba = 0; // tu jest zapisywana typowana liczba

  public void zgaduj() {
    liczba = (int) (Math.random() * 10);
    System.out.println("Typuję liczbę: " + liczba);
  }
}

class StarterGry {
  public static void main(String[] args) {
    Zgadywanka gra = new Zgadywanka();
    gra.rozpocznijGre();
  }
}
```

Wyniki (za każdym razem będą inne)



Java sama wynosi śmieci

Za każdym razem, gdy w Javie jest tworzony obiekt, trafia on do obszaru pamięci nazywanego **stertą**. Wszystkie obiekty, niezależnie od tego, kiedy, jak i gdzie zostaną utworzone, zawsze są przechowywane na stercie. Jednak nie jest to sterta starej, zapomnianej pamięci; w Javie sterta jest także nazywana **stertą automatycznie odśmiecaną**. Kiedy tworzysz obiekt, Java rezerwuje na stercie obszar o wielkości odpowiadającej potrzebom konkretnego obiektu. Obiekt posiadający, dajmy na to, 15 składowych, będzie prawdopodobnie potrzebował więcej miejsca niż obiekt mający jedynie 2 składowe. Co się jednak dzieje, kiedy będzie trzeba odzyskać miejsce przydzielone na stercie? W jaki sposób można usunąć z niej obiekt, kiedy skończymy go już używać? To Java zarządza pamięcią za nas! Kiedy wirtualna maszyna Javy (JVM) „zauważy", że obiekt nie będzie już mógł być wykorzystywany w programie, zostaje on uznany za *nadający się do odśmiecenia*, a kiedy w systemie zacznie brakować pamięci, zostanie uruchomiony odśmiecacz, który usunie z niej wszystkie nieosiągalne obiekty. W ten sposób pamięć zostanie zwolniona i będzie można ją ponownie wykorzystać. Więcej informacji na ten temat znajdziesz w kolejnym rozdziale.

Nie istnieją głupie pytania

P: Co zrobić, jeśli będę potrzebować globalnych zmiennych i metod? Jak to zrobić, jeśli wszystko musi być umieszczane wewnątrz klas?

O: W programach obiektowych pisanych w Javie nie istnieje pojęcie zmiennych lub metod „globalnych". W szczególnych zastosowaniach istnieją jednak sytuacje, gdy chcemy, aby metoda (lub stała) była dostępna dla dowolnego fragmentu kodu działającego w dowolnej części programu. Przypomnij sobie metodę `random()` zastosowaną w programie krasomówczym, stanowi ona doskonały przykład metody, którą można wywołać w dowolnym miejscu programu. Albo, na przykład, stała *pi*. W rozdziale 10. dowiesz się, że oznaczenie metod jako publiczne (przy użyciu słowa kluczowego `public`) i statyczne (przy użyciu słowa kluczowego `static`) sprawia, że zachowują się one jak metody „globalne" — będzie miał do nich dostęp dowolny kod działający w dowolnej klasie wchodzącej w skład programu. Z kolei, jeśli zmienna zostanie oznaczona jako publiczna, statyczna i finalna (odpowiednio przy użyciu słów kluczowych: `public`, `static` oraz `final`), to w efekcie stanie się ona globalnie dostępną stałą.

P: Ale co to za obiektowość, skoro wciąż można tworzyć zarówno funkcje, jak i dane globalne?

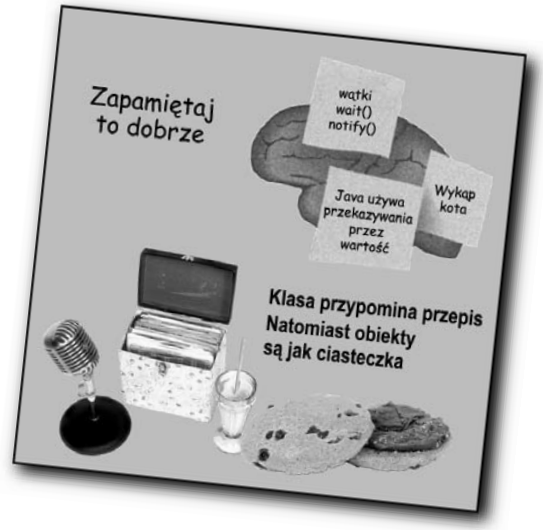
O: Przede wszystkim wszystko, co jest tworzone w Javie, musi być umieszczone w jakiejś klasie. Zatem stała *pi* oraz metoda `random()`, choć statyczne i publiczne, to jednak są zdefiniowane w klasie `Math`. Poza tym należy pamiętać, że takie dane i metody „globalne", stanową w Javie raczej wyjątek, a nie regułę. Stanowią one szczególny przypadek, w którym nie trzeba tworzyć wielu kopii obiektu, aby skorzystać z jego danych lub metod.

P: Czym jest program pisany w Javie? Co się w zasadzie rozpowszechnia?

O: Program napisany w Javie to grupa klas (a przynajmniej *jedna* klasa). W każdej aplikacji dokładnie jedna klasa musi mieć metodę `main()`, która służy do uruchamiania programu. A zatem Ty, jako programista, tworzysz jedną lub większą ilość klas. I właśnie one są rozpowszechniane jako program. Jeśli użytkownik końcowy nie posiada JVM, to do klas tworzących aplikację trzeba będzie dołączyć także środowisko wykonawcze Javy, dzięki któremu użytkownicy będą mogli uruchomić program. Dostępnych jest wiele programów instalacyjnych pozwalających na łączenie własnych klas z wieloma różnymi wersjami JVM (na przykład w zależności od docelowej platformy systemowej) i zapisywanie wszystkich niezbędnych plików na płycie CD-ROM. W ten sposób użytkownik końcowy może zainstalować odpowiednią wersję JVM (zakładając, że na jego komputerze wirtualna maszyna Javy nie jest jeszcze zainstalowana).

P: A co w sytuacji, gdy moją aplikację tworzy sto klas? Albo tysiąc? Czy dostarczanie tylu plików nie jest poważnym utrudnieniem? Czy nie można z nich zrobić jednego dużego, wykonywalnego pliku aplikacji?

O: Owszem, dostarczanie użytkownikowi końcowemu tak dużej ilości plików byłoby kłopotliwe. Na szczęście nie jest to konieczne. Można umieścić wszystkie pliki tworzące aplikację w jednym „archiwum Javy" — pliku *.jar* — bazującym na formacie archiwów pkzip. W pliku jar można umieścić odpowiednio sformatowany plik tekstowy stanowiący tak zwany *manifest* i określający, która klasa umieszczona w danym archiwum zawiera metodę `main()`, którą należy wywołać.



CELNE SPOSTRZEŻENIA

- Programowanie obiektowe pozwala na rozszerzanie programów bez konieczności modyfikowania przetestowanego wcześniej działającego kodu.
- W Javie cały tworzony kod jest umieszczany wewnątrz **klas**.
- Klasa opisuje, jak należy tworzyć obiekty danego typu. **Można ją zatem porównać do wzorca.**
- Obiekt potrafi o siebie zadbać; nie musisz ani wiedzieć, ani zaprzątać sobie głowy tym, *jak* obiekt coś robi.
- Obiekt **posiada** informacje i **wykonuje** czynności.
- Informacje, jakie obiekt ma na swój temat, są przechowywane w tak zwanych **składowych**. Reprezentują one *stan* danego obiektu.
- Czynności, jakie obiekt wykonuje, są nazywane **metodami**. Określają one *działanie* (lub *zachowanie*) obiektu.
- Tworząc klasę, można także stworzyć niezależną klasę testową służącą do tworzenia i sprawdzania obiektów nowej klasy.
- Klasa może **dziedziczyć** składowe i metody po bardziej abstrakcyjnych klasach **bazowych**.
- W czasie wykonywania program Javy jest w zasadzie grupą wzajemnie komunikujących się obiektów.

Ćwiczenie

BĄDŹ kompilatorem

Każdy z plików przedstawionych na tej stronie stanowi niezależny kompletny plik źródłowy. Twoim zadaniem jest stać się kompilatorem i określić, czy przedstawione programy skompilują się czy nie. Jeśli nie można ich skompilować, to jak je poprawić? Jeśli można je skompilować, to jakie wygenerują wyniki?

A

```
class Magnetofon {
  boolean mozeNagrywac = false;

  void odtworzTasme() {
    System.out.println("odtwarzam taśmę");
  }

  void nagrajTasme() {
    System.out.println("nagrywam taśmę");
  }
}

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

    m.mozeNagrywac = true;
    m.odtworzTasme();

    if (m.mozeNagrywac == true) {
      m.nagrajTasme();
    }
  }
}
```

B

```
class OdtwarzaczDVD {
  boolean mozeNagrywac = false;

  void nagrajPlyteDVD() {
    System.out.println("nagrywam płytę DVD");
  }
}

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

    OdtwarzaczDVD o = new OdtwarzaczDVD();
    o.mozeNagrywac = true;
    o.odtworzPlyteDVD();

    if (o.mozeNagrywac == true) {
      o.nagrajPlyteDVD();
    }
  }
}
```

Ćwiczenie

Magnesiki z kodem

Działający program Javy został podzielony na fragmenty, zapisany na małych magnesach, które przyczepiono do lodówki. Czy jesteś w stanie złożyć go z powrotem w jedną całość, tak aby wygenerował przedstawione poniżej wyniki? Niektóre nawiasy klamrowe spadły na podłogę i były zbyt małe, aby można je było podnieść; dlatego w razie potrzeby możesz je dodawać!

```
p.zagrajNaBebnie();
```

```
Perkusja p = new Perkusja();
```

```
boolean talerze = true;
boolean beben = true;
```

```
void zagrajNaBebnie() {
  System.out.println("bam, bam, baaaa-am-am");
}
```

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

```
if (p.beben == true) {
  p.zagrajNaBebnie();
}
```

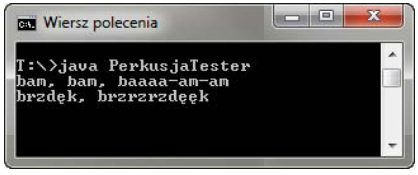
```
p.beben = false;
```

```
class PerkusjaTester {
```

```
p.zagrajNaTalerzach();
```

```
class Perkusja {
```

```
void zagrajNaTalerzach() {
  System.out.println("brzdęk, brzrzrzdęęk");
}
```



Zagadkowy basen

Twoim ***zadaniem*** jest wybranie fragmentów kodu z basenu i umieszczenie ich w miejscach kodu oznaczonych podkreśleniami. Każdy fragment kodu **możne** być użyty więcej niż raz, a co więcej, nie wszystkie fragmenty zostaną wykorzystane. ***Zadanie*** polega na stworzeniu klasy, którą będzie można skompilować i która wygeneruje wyniki przedstawione poniżej. Nie daj się zwieść pozorom — ta zagadka jest trudniejsza, niż można by przypuszczać.

```
public class EchoTester {
  public static void main(String[] args) {
    Echo e1 = new Echo();
    _____________________________________
    int x = 0;
    while ( _____ ) {
      e1.witaj();
      ____________________
      if ( _______ ) {
        e2.ilosc = e2.ilosc + 1;
      }
      if ( ______ ) {
        e2.ilosc = e2.ilosc + e1.ilosc;
      }
      x = x + 1;
    }
    System.out.println(e2.ilosc);
  }
}
```

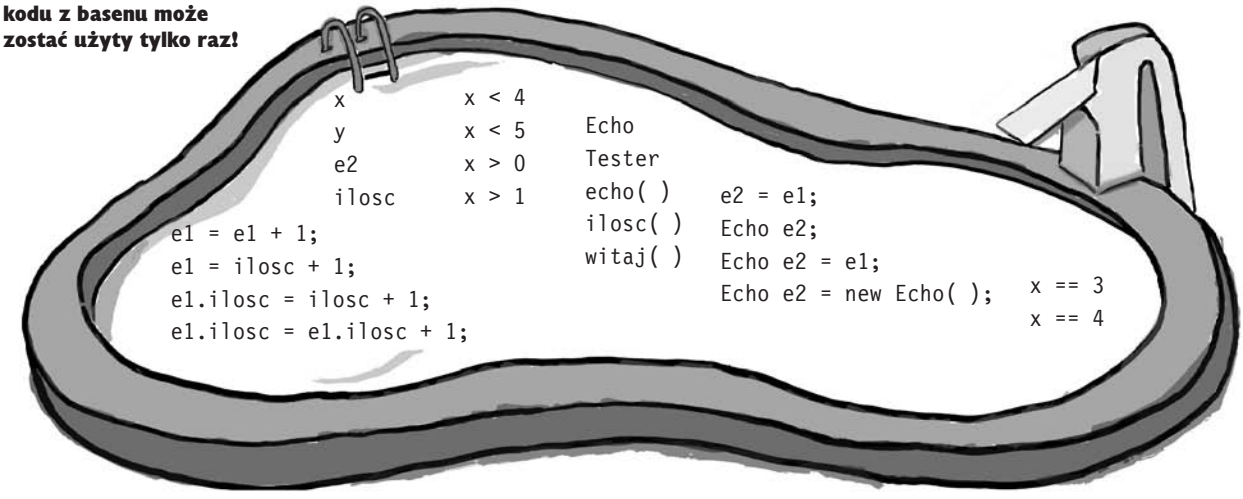
Wyniki:


```
class _____ {
  int ______ = 0;
  void _____ {
    System.out.println("sieeeeemasz... ");
  }
}
```

Pytanie dodatkowe!

Jak rozwiązałbyś zagadkę, gdyby w ostatnim wierszu wyników pojawiła się liczba **24**, a nie **10**?

Notatka: Każdy fragment kodu z basenu może zostać użyty tylko raz!



Kim jestem?

Grupa zamaskowanych komponentów Javy gra w grę towarzyską o nazwie „Zgadnij, kim jestem?". Komponenty dają Ci podpowiedzi, a Ty na ich podstawie starasz się odgadnąć, kim one są. Załóż, że komponenty zawsze mówią prawdę. Jeśli mówią coś, co może być prawdą w odniesieniu do kilku z nich, wybierz wszystkie komponenty, dla których podane stwierdzenie jest prawdziwe. W pustych miejscach obok podanych podpowiedzi podaj nazwy komponentów biorących udział w zabawie. Odpowiedź na pierwszą podpowiedź podaliśmy sami.

W dzisiejszej zabawie udział biorą:

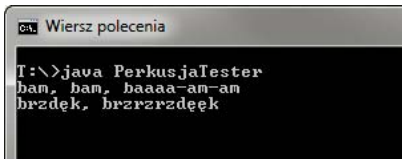
Klasa Metoda Obiekt Składowa

Jestem kompilowana na podstawie pliku .java.	*klasa*
Wartości moich składowych mogą się różnić od wartości składowych mojego brata bliźniaka.	
Działam jak wzorzec.	
Lubię działać.	
Mogę mieć wiele metod.	
Reprezentuję „stan".	
Mam swoje „działanie".	
Przebywam w obiektach.	
Istnieję na stercie.	
Służę do tworzenia kopii obiektów.	
Mój stan może się zmieniać.	
Deklaruję metody.	
Mogę się zmieniać w trakcie działania programu.	

Ćwiczenie rozwiązanie

Magnesiki z kodem

```
class Perkusja {

  boolean talerze = true;
  boolean beben = true;

  void zagrajNaBebnie() {
    System.out.println("bam, bam, baaaa-am-am");
  }

  void zagrajNaTalerzach() {
    System.out.println("brzdęk, brzrzrzdęęk");
  }
}

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

    Perkusja p = new Perkusja();
    p.zagrajNaBebnie();
    p.beben = false;
    p.zagrajNaTalerzach();

    if (p.beben == true) {
      p.zagrajNaBebnie();
    }
  }
}
```



Bądź kompilatorem

```
class Magnetofon {
  boolean mozeNagrywac = false;

  void odtworzTasme() {
    System.out.println("odtwarzam taśmę");
  }

  void nagrajTasme() {
    System.out.println("nagrywam taśmę");
  }
}
```

A

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

    Magnetofon m = new Magnetofon();
    m.mozeNagrywac = true;
    m.odtworzTasme();

    if (m.mozeNagrywac == true) {
      m.nagrajTasme();
    }
  }
}
```

Mamy już wzorzec, teraz musimy stworzyć obiekt!

```
class OdtwarzaczDVD {
  boolean mozeNagrywac = false;
  void nagrajPlyteDVD() {
    System.out.println("nagrywam płytę DVD");
  }
  void odtworzPlyteDVD() {
    System.out.println("odtwarzam płytę DVD");
  }
}
```

B

```
class OdtwarzaczDVDTester {
  public static void main(String[] args) {
    OdtwarzaczDVD o = new OdtwarzaczDVD();
    o.mozeNagrywac = true;
    o.odtworzPlyteDVD();
    if (o.mozeNagrywac == true) {
      o.nagrajPlyteDVD();
    }
  }
}
```

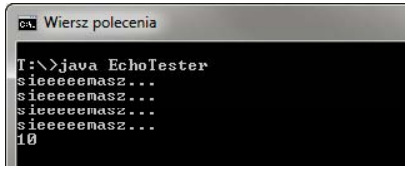
Wiersz kodu zawierający wywołanie o.odtworzPlyteDVD() nie skompiluje się, jeśli metoda nie będzie istnieć.

Rozwiązanie zagadki

Zagadkowy basen

```
public class EchoTester {
  public static void main(String[] args) {
    Echo e1 = new Echo();
    Echo e2 = new Echo(); // poprawna odpowiedź
    // -- lub --
    Echo e2 = e1; // odpowiedź na pytanie dodatkowe
    int x = 0;
    while ( x < 4 ) {
      e1.witaj();
      e1.ilosc = e1.ilosc + 1;
      if ( x == 3 ) {
        e2.ilosc = e2.ilosc + 1;
      }
      if ( x > 0 ) {
        e2.ilosc = e2.ilosc + e1.ilosc;
      }
      x = x + 1;
    }
    System.out.println(e2.ilosc);
  }
}
```

```
class Echo {
  int ilosc = 0;
  void witaj() {
    System.out.println("sieeeeemasz... ");
  }
}
```



Kim jestem?

Jestem kompilowana na podstawie pliku .java.	*klasa*
Wartości moich składowych mogą się różnić od wartości składowych mojego brata bliźniaka.	*obiekt*
Działam jak wzorzec.	*klasa*
Lubię działać.	*obiekt, metoda*
Mogę mieć wiele metod.	*klasa, obiekt*
Reprezentuję „stan".	*składowa*
Mam swoje „działanie".	*obiekt, klasa*
Przebywam w obiektach.	*metoda, składowa*
Istnieję na stercie.	*obiekt*
Służę do tworzenia kopii obiektów.	*klasa*
Mój stan może się zmieniać.	*obiekt, składowa*
Deklaruję metody.	*klasa*
Mogę się zmieniać w trakcie działania programu.	*obiekt, składowa*

Notatka: Mówi się, że zarówno klasy, jak i obiekty mają stan i działanie. Są one co prawda definiowane w klasie, jednak mówimy, że także obiekty je „posiadają". Jak na razie nie obchodzi nas, gdzie się one znajdują z technicznego punktu widzenia.