Jacek Matulewski
http://www.fizyka.umk.pl/~jacek/

Java w NetBeans
część 2: Język Java

Skrypt dla słuchaczy studium podyplomowego SPIN (WFAiIS UMK)

Toruń, 9 listopada 2007

Najnowsza wersja skryptu i źródła jest dostępna pod adresem
http://www.fizyka.umk.pl/~jacek/dydaktyka/java_spin/

Source level 1.1/1.2

Spis treści

Kilka słów wstępu

Programistom C/C++ Java powinna wydawać się znajoma. Odnajdą w niej większość słów kluczowych, które stosowali w C/C++, choć nie zawsze mechanizm, jaki się za nimi kryje, jest tym samym, który znamy z C/C++. Odnajdziemy również podobieństwa do Object Pascala. Java to język, który największą karierę zrobił w sieci. Dlaczego? Spełnia podstawowy warunek niewielkich rozmiarów skompilowanych apletów. Poza tym w język wbudowane są systemy zabezpieczeń chroniące komputer-klient przed atakiem z wykorzystaniem apletów. Również po stronie serwera Java jest często wykorzystywanym narzędziem szczególnie ze względu na możliwości obsługi baz danych. To wszystko razem powoduje, że Java to język z perspektywami.

Nowe i „stare" cechy Javy

Java jako język programowania niezwykle przypomina C++. To nie jest oczywiście przypadek — Java ma być czymś naturalnym dla programistów C++, a jednocześnie pozbawionym niebezpieczeństw i niedogodności C++. Nie ma zatem wskaźników, uporządkowane jest zarządzanie pamięcią, tablice są obiektami z wbudowanymi zabezpieczeniami przed dostępem do elementów spoza tablicy, wygodny, i co najważniejsze jeden, typ łańcuchowy, wygodny sposób tworzenia dodatkowych wątków, konsekwentne wykorzystanie wyjątków do obsługi błędów, konsekwentna obiektowość bez możliwości definiowania funkcji niebędących metodami oraz zmiennych globalnych, możliwość tworzenia klas wewnętrznych w innych klasach, interfejsy oraz jedna standardowa biblioteka klas, która jest integralną częścią języka. To ostatnie zasługuje na szczególną uwagę.

Zwykle dotąd języki programowania oznaczały jedynie „gramatykę". Programista mógł wykorzystywać biblioteki dostarczane przez niezależnych producentów[1], ale wówczas musiał liczyć się z koniecznością dołączenia ich do dystrybucji swojego programu. W przypadku Javy bogaty zestaw komponentów umożliwiających zbudowanie interfejsu apletu lub aplikacji, narzędzia do programowania współbieżnego, sieciowego, klasy służące do tworzenia grafiki oraz wiele innych instalowane są razem z JDK. Bogaty zestaw bibliotek znajduje się więc na każdym komputerze, na którym w ogóle można uruchomić aplet lub aplikację Javy. W tej sytuacji nie ma oczywiście najmniejszej potrzeby rozpowszechniania go razem z apletem lub aplikacją, co znakomicie ułatwia proces dystrybucji programów.

Pomimo tych wszystkich nowości praktyka programowania, jeżeli można się tak wyrazić, pozostaje taka sama jak w C++. Filozofia programowania obiektowego, konstrukcje sterowania przepływem, korzystanie z tablic — to wszystko można niemal kopiować z kodów C++.

Wirtualna maszyna Javy. Przenośność

Charakterystyczną cechą Javy jest sposób kompilacji oraz uruchamiania apletów i aplikacji. Kompilacja nie sprowadza kodu źródłowego do kodu maszynowego, czyli do postaci zrozumiałej dla procesora, ale do tzw. *bytecode*. Ten ostatni jest zrozumiały tylko dla wirtualnej maszyny Javy (JVM z ang. *Java Virtual Machine*), która interpretując go wydaje procesorowi kolejne polecenia.

Tym wszystkim oczywiście nie musimy się martwić. Należy tylko wiedzieć, jak przy braku plików *.exe* lub *.com* uruchamiać programy napisane w Javie i to oczywiście będzie wyjaśnione w kolejnych rozdziałach.

Są dwie zasadnicze zalety takiego sposobu kompilacji i uruchamiania. Po pierwsze, wirtualna maszyna Javy oddziela program od systemu wprowadzając dodatkową warstwę chroniącą system przed niestabilnością programów oraz dostarczającą potrzebne biblioteki. Druga zaleta to możliwość przenaszalności nie tylko kodu źródłowego, jak było (teoretycznie) w C++, ale również skompilowanych programów. Pliki ze skompilowanym apletem lub aplikacją są rozumiane przez wszystkie wirtualne maszyny Javy bez względu na system, w którym są one zainstalowane. I to wirtualna maszyna Javy odpowiada za komunikację z tym systemem, bez względu na to, czy mamy do czynienia z Linuksem, Solaris czy Windows.

[1] Np. biblioteka VCL dodawana przez Borland do C++Builder i Delphi.

Jak zwykle wadą takiego rozwiązania jest oczywiście to, co jest też jego zaletą, a więc istnienie warstwy pośredniczącej i względnie wolne interpretowanie *bytecode*. Na szczęście i na to znalazła się rada w postaci kompilatora typu *Just-In-Time* (JIT), który zamienia od razu cały *bytecode* w kod maszynowy. Kompilatory tego typu są elementami wirtualnej maszyny Javy od JDK w wersji 1.1, co znacznie zwiększyło szybkość wykonywania programów.

Java i JVM jest pierwowzorem kolejnych rozwiązań tego samego typu. Mam na myśli Microsoft .NET Framework i język C#. Podobieństwa rozwiązań technicznych (warstwa pośrednia i sposób kompilacji) oraz podobieństwa samych języków Java i C# są bardzo duże. Jest to oczywiście ogromna zaleta dla uczących się programowania osób, ponieważ poznając jeden z języków z rodziny C++, Java, C# uczymy się jakby ich wszystkich.

Język

Poniżej postaram się przybliżyć Czytelnikowi język Java. Nie będzie to oczywiście wyczerpujące przedstawienie gramatyki języka ani tym bardziej omówienie jego wbudowanych bibliotek. Tu skupimy się na podstawowych elementach: typach danych, instrukcjach sterowania przepływem, obsłudze łańcuchów i tablic tylko w takim zakresie, jaki jest konieczny do zrozumienia ćwiczeń z tej książki[2].

Przegląd ma taką formę, że po pierwszym jego przeczytaniu powinien dalej służyć jako leksykon wyjaśniający wątpliwości, które mogą się pojawić przy okazji wykonywania ćwiczeń z pozostałych rozdziałów. Nie ma w nim, poza częścią dotyczącą instrukcji sterujących, ćwiczeń.

Komentarze

Java dopuszcza dwa typy komentarzy znanych z C++:

```
//to jest linia ignorowana przez kompilator
int i=1; //ta część linii również jest ignorowana przez kompilator


/*
to jest wiele linii
ignorowanych
przez kompilator
*/
```

W pierwszym ignorowane są wszystkie znaki od sekwencji `//` do końca linii. W drugim wszystko, co znajduje się między `/*` i `*/`. Ponadto w Javie jest jeszcze jeden typ komentarzy, które mogą być wykorzystywane do automatycznego tworzenia dokumentacji:

```
/**
te linie również są
ignorowane przez kompilator,
ale mogą być wykorzystane
do automatycznego tworzenia
dokumentacji
*/
```

lub

```
/**
 * dodatkowe gwiazdki dodają
```

[2] Książek, które pozwalają na poznanie Javy, jest wiele. Na początek proponuję książkę Marcina Lisa *Java. Ćwiczenia praktyczne*, Wydawnictwo Helion, Gliwice 2002.

```
 * jedynie elegancji komentarzom
 */
```

Szczegóły dotyczące wykorzystania tego typu komentarzy znajdują się w rozdziale 7.

Konwencja wielkości liter

W Javie wielkość liter ma znaczenie. Słowa kluczowe, nazwy zmiennych i klas powinny mieć dokładnie taką pisownię jak w ich definicjach i jaka została podana w dokumentacji. Istnieje konwencja używania wielkich liter, która nie ma oczywiście znaczenia z punktu widzenia kompilatora, ale na pewno pomaga szybciej zrozumieć kod.

- Zgodnie z tą konwencją nazwy klas piszemy z dużej litery. Jeżeli nazwa klasy składa się z wielu połączonych wyrazów, każdy wyraz piszemy z dużej litery, np. `PierwszyAplet`.
- Nazwy zmiennych, obiektów, metod i właściwości piszemy z małej litery, ale kolejne wyrazy w wielowyrazowych nazwach pisane są z dużej litery, np. obiekt `pierwszyAplet1`, metoda `pierwszaMetoda` itd.
- Nazwy pakietów piszemy małymi literami w całości. Bez względu na to, czy są skrótami, czy też składają się z wielu wyrazów, np. `java.awt, java.awt.datatransfer`.
- Obiekty będące statycznymi właściwościami klas pisane są czasami drukowanymi literami, np. `Color.WHITE`.

Proste typy danych

Dostępne typy danych niebędące klasami zebrane zostały w tabeli 2.1.

Tabela 2.1. Dostępne w Javie proste typy danych oraz ich zakresy

Nazwa typu (słowo kluczowe)	Ilość zajmowanych bitów	Możliwe wartości (zakres, włącznie)
`boolean`		true, false
`byte`	8 bitów = 1 na znak, 7 wartość	od –128 do 127
`short`	2 bajty = 16 bitów	od –32768 do 32767
`int`	4 bajty = 32 bity	od –2147483648 do 2147483647
`long`	8 bajtów = 64 bity	od –9223372036854775808 do 9223372036854775807
`char`	2 bajty = 16 bitów (zobacz komentarz)	od 0 do 65535
`float`	4 bajty (zapis zmiennoprzecinkowy)	Wartość może być dodatnia lub ujemna. Najmniejsza bezwzględna wartość to $1.4 \cdot 10^{-45}$, największa to ok. $3.403 \cdot 10^{38}$
`double`	8 bajtów (zapis zmiennoprzecinkowy)	Najmniejsza wartość to $4.9 \cdot 10^{-324}$, największa to ok. $1.798 \cdot 10^{308}$

Proszę zwrócić uwagę, że typ `char` jest kodowany przez 2 bajty. Dwa bajty pozwalają na przechowywanie znaków Unicode w zmiennych typu `char`. W ten sposób Java unika też problemów z ograniczeniem do 256 znaków znanym np. z C++ oraz rozwiązuje w sposób ogólny problem z kodowaniem znaków specyficznych dla różnych języków (włączając w to całe alfabety, jak cyrylica i alfabet arabski).

Operatory arytmetyczne

Na prostych typach danych można wykonywać wszystkie znane ze szkoły podstawowej operacje: dodawanie, odejmowanie, mnożenie (znak `*`) oraz dzielenie (znak `/`). Zdefiniowane są również operatory dodatkowe:

Zamiast często stosowanej operacji zwiększania wartości zmiennej o zadaną wartość `c=c+2;` można użyć operatora `+=`: `c+=2;`. Jeżeli wartość zmieniana jest o jeden, to możemy napisać `c++;`. Ta ostatnia postać jest często wykorzystywana przy konstruowaniu pętli. Analogiczne postacie mogą mieć operacje odejmowania. Dla operacji mnożenia i dzielenia zdefiniowane są tylko operatory `*=` i `/=`.

Konwersje typów

Operatory mogą mieć argumenty różnego typu. Wówczas dokonywana jest niejawna konwersja typów np.

```
double a=0.0;
int b=0;
double d=a+b;
```

Zwracana przez operator `+` wartość będzie w tym przypadku typu `double`. Zawsze wybierany jest typ bardziej dokładny lub o większym zakresie.

Niejawna konwersja jest zawsze możliwa z typu naturalnego w zmiennoprzecinkowy oraz z typu o mniejszym zakresie na typ o większym zakresie (kodowany przez większą liczbę bitów). Dlatego przy próbie przypisania wartości typu `double` do zmiennej typu `int` pojawi się błąd kompilatora ostrzegający przed możliwą utratą precyzji:

```
int c=a+b; //tu wystapi blad kompilatora (możliwa utrata dokładności)
```

Możemy wymusić takie przypisanie, ale wówczas jawnie musimy dodać operator konwersji `(typ)`, np.

```
int c=(int)(a+b);
```

Należy sobie jednak zdawać sprawę, że liczba zostanie wówczas zaokrąglona w dół, np. wartość wyrażenia `(int)0.99` to zero, a nie jeden. Jeżeli nie jest to zgodne z naszym oczekiwaniem, należy rozważyć stosowanie metod zaokrąglających z klasy `Math`.

Uwaga

Operacja dzielenia dla argumentów będących liczbami całkowitymi daje w wyniku liczbę całkowitą. Jeżeli jest to konieczne, wynik jest zaokrąglany do liczby całkowitej w dół. Zgodnie z tym operacje `1/2` i `3/4` zwrócą wartość `0`. Jeżeli chcemy uzyskać prawdziwą wartość wynikającą z arytmetyki, powinniśmy napisać `1/(double)2` lub `1/2.0`. W obu przypadkach mianownik traktowany jest tak jak liczba typu `double`, więc wykorzystywany jest przeciążony operator `/` dla argumentów typu `double` zwracający wartość typu `double`.

Klasy opakowujące

Każdy z podstawowych typów w Javie ma klasę opakowującą. na przykład typ `double` ma klasę opakowującą o nazwie `Double`. Zadaniem klas opakowujących jest dostarczenie podstawowych informacji o typie (np. o zakresie informują pola `MIN_VALUE`, `MAX_VALUE` i `TYPE`) oraz metod służących do jawnej konwersji. W przypadku konwersji z liczb zmiennoprzecinkowych do całkowitych jest to konwersja zawsze z zaokrągleniem w dół.

Klasy opakowujące są szczególnie często wykorzystywane do konwersji do i z łańcuchów. Nawet konstruktory każdej z nich są nawet przeciążone w taki sposób, żeby przyjmować argument w postaci łańcucha.

```
Double g=new Double("3.14"); //konwersja z łańcucha do obiektu typu Double
Double h=Double.valueOf("3.14"); //j.w.
double i=Double.parseDouble("3.14"); //konwersja z łańcucha do typu prostego double
String s=g.toString(); //konwersja obiektu Double do łańcucha
String t=Double.toString(3.14); //konwersja liczby double do łańcucha
```

W trzecim i ostatnim przypadku nie powstają obiekty klasy opakowującej. Do konwersji między łańcuchem, a typem prostym. Wykorzystywane są jej metody statyczne. Analogiczne metody mają wszystkie klasy opakowujące `Float`, `Integer`, `Long` itd.

Może zdziwić fakt, że nie jest możliwa niejawna ani nawet jawna konwersja z klasy opakowującej do typu prostego. Trzeba skorzystać z odpowiedniej metody klasy opakowującej:

```
Double j=new Double(Math.PI);
double k=(double)j; //tu pojawi się blad
double k=j.doubleValue();
```

Łańcuchy

Z góry zastrzegam, że pełne zrozumienie niektórych aspektów korzystania z łańcuchów w Javie będzie możliwe dopiero po zapoznaniu się z ogólnymi właściwościami klas i obiektów, bo w Javie łańcuchy są właśnie obiektami. Jednak jednocześnie łańcuchy są podstawowym elementem każdego języka, który jest wykorzystywany bardzo często.

Do obsługi łańcuchów służy klasa `String`. I tu muszę z ulgą westchnąć, bo jest to chyba pierwszy język, w którym sprawę łańcuchów rozwiązano w sposób prosty, jednolity i wygodny. Obiekt klasy `String` można stworzyć korzystając z konstruktora lub inicjując łańcuchem zapisanym w podwójnych cudzysłowach:

```
String s1="Helion";
String s2=new String("Helion");
```

Oba sposoby są zupełnie równoważne, tzn. napis `"Helion"` jest pełnoprawnym obiektem i można wywoływać jego metody, np.

```
int l1="Helion".length();
```

zupełnie tak samo jak w przypadku referencji do łańcucha:

```
int l2=s2.length();
```

Zmienne `s1` i `s2` są referencjami do łańcuchów, tzn. reprezentują obiekty typu `String`. Zadeklarowanie referencji

```
String s0;
```

nie oznacza jeszcze zainicjowania obiektu[3]. Obiekt nie został w tym przypadku utworzony, a więc referencja nie przechowuje żadnej wartości. Dobrym zwyczajem jest inicjowanie referencji, nawet jeżeli nie tworzymy równocześnie obiektów:

```
String s0=null;
```

Poza konstruktorem przyjmującym w argumencie wartość łańcucha w klasie `String` zdefiniowanych jest wiele innych konstruktorów konwertujących do łańcucha najróżniejsze typy danych. Najważniejszym jest konstruktor przyjmujący w argumencie tablicę znaków znaną z C/C++.

Dozwolone są wszystkie wygodne operacje łączenia łańcuchów[4]:

```
String s3="Hel"+"ion";
s3+=", Gliwice";
```

Co więcej, operator `+` jest przeciążony w taki sposób, że możliwe jest automatyczne skonwertowanie do łańcucha wyrażeń stojących po prawej stronie operatora `+`, jeżeli lewy argument jest łańcuchem:

```
String s4=s3+" "+44+"-"+100; //"Helion, Gliwice 44-100"
String s5="Pi: "+Math.PI;    //"Pi: 3.141592653589793"
String s6=""+Math.sqrt(2);   //"1.4142135623730951"
String s7=""+Color.white;    //"java.awt.Color[r=255,g=255,b=255]"
```

Konwersja z wykorzystaniem łańcucha pustego użyta w dwóch ostatnich przykładach jest często wykorzystywanym trikiem, który ułatwia zamianę różnego typu zmiennych i obiektów w łańcuchy. Zmiana w

[3] Omówienie referencji nastąpi później. Na razie należy wiedzieć, że referencja reprezentuje obiekt, jest odnośnikiem do niego.

[4] Ściśle rzecz biorąc, są to operacje na referencjach, których skutkiem jest łączenie łańcuchów. Referencje to nie to samo co obiekty – szczegółowo omówię ten temat przy wstępie do programowania obiektowego.

ten sposób obiektu w łańcuch jest możliwa dzięki automatycznemu wywołaniu metody `toString` zdefiniowanej w klasie `java.lang.Object`, a więc obecnej w każdym obiekcie[5].

Konwersje do łańcucha z typów prostych można również wykonywać za pomocą wielokrotnie przeciążonej metody `String.valueOf` np.

```
String s8=s3+" "+String.valueOf(44)+"-"+String.valueOf(100);
String s9="Pi: "+String.valueOf(Math.PI);
String s10=""+String.valueOf(Math.sqrt(2));
String s11=""+String.valueOf(Color.white);
```

Efekt jest identyczny jak przy niejawnej konwersji podczas korzystania z operatora `+`.

W tabeli 2.2 przedstawiono kilka pożytecznych metod klasy `String`. Jak widać z tabeli, możliwości klasy są spore. Bez problemu wystarczają w najczęstszych zastosowaniach.

Tabela 2.2. Najczęściej stosowane metody klasy String

Funkcja wartość **nazwa**(argumenty)	Opis	Przykład zastosowania
`char` **`charAt`**`(int)`	Zwraca znak na wskazanej pozycji. Pozycja pierwszego znaku to 0	`"Helion".charAt(0)`
`boolean` **`equals`**`(Object)`	Porównuje łańcuch z podanym w argumencie	`"Gliwice".equals("Toruń")`
`int` **`compareTo`**`(String),` **`compareToIgnoreCase`**	Porównuje łańcuchy. Zwraca 0, jeżeli równe i liczby większe lub mniejsze od zera przy różnych. Metoda przydatna przy sortowaniu i przeszukiwaniu.	
`int` **`indexOf`**`(char)` `int` **`indexOf`**`(String),`	Pierwsze położenie litery lub łańcucha znalezione w łańcuchu, na rzecz którego wywołana została metoda	`"Helion".indexOf('e')` `"Helion".indexOf("eli")` `"Bagabaga".indexOf("ag")`
`int` **`lastIndexOf`**`(char)` `int` **`lastIndexOf`**`(String)`	j.w., ale ostatnie wystąpienie litery lub łańcucha	`"Bagabaga".lastIndexOf("ag")`
`int` **`length`**`()`	Zwraca długość łańcucha	`"Helion".length()`
`replaceFirst`, **`replaceAll`**	Zamiana wskazanego fragmentu przez inny	`"Helion".replaceFirst("lion","aven")`
`String` **`substring`**`(int,int)`	Fragment łańcucha pomiędzy wskazanymi pozycjami (druga liczba oznacza pozycję niewchodzącą już do wybranego fragmentu)	`"Helion".substring(2,6)`
`toLoweCase`, **`toUpperCase`**	Zmienia wielkość wszystkich liter	`"Helion".toUpperCase()`
`String` **`trim`**`()`	Usuwa spacje z przodu i z tyłu łańcucha	`"   Helion   ".trim()`

Ciągi definiujące łańcuchy mogą zawierać sekwencje specjalne rozpoczynające się od znaku backslash `\` (identycznie jak w C/C++). Często wykorzystywany jest znak końca linii `\n` oraz znak cudzysłowu `\"` (ten ostatni nie kończy łańcucha, jest traktowany dokładnie tak samo jak inne znaki). Sekwencja kasująca poprzedni znak to `\b`. Wreszcie sekwencje pozwalające definiować znaki spoza dostępnego na klawiaturze zestawu ASCII zaczynają się od `\u` i obejmują cztery kolejne znaki alfanumeryczne[6], np. `\u0048`.

[5] Więcej na temat możliwości konwertowania w ten sposób obiektów dalszej części książki.

[6] Przypominam, że w Javie typ char jest dwubajtowy (zobacz tabela typów prostych). Trzeba go zatem kodować aż czterocyfrowymi liczbami szesnastkowymi. Pod Windows właściwe liczby Unicode można znaleźć w aplikacji `Tablica` znaków w menu *Start/Akcesoria/Narzędzia systemowe*.

```
String helion="Wydawnictwo \" \u0048 \u0065 \u006c \u0069 \u006f \u006e \"";
```

Wartość tego łańcucha to „Wydawnictwo " H e l i o n " ” (spacje w jego definicji zostały umieszczone tylko po to, żeby wyraźniej pokazać sekwencje dla każdego znaku Unicode).

Ze względu na wykorzystanie znaku backslash `\` do rozpoczynania sekwencji kodujących, również dla wprowadzenia tego znaku konieczna jest specjalna sekwencja `\\`. Stąd biorą się podwójne znaki backslash w C/C++ i Javie w przypadku zapisanych w łańcuchach ścieżek dostępu do plików w Windows:

```
String nazwapliku="C:\\Windows\\NOTEPAD.EXE";
```

Wartość tego łańcucha to w rzeczywistości `C:\Windows\NOTEPAD.EXE`.

Tablice

Definicja tablicy w dowolnym języku programowania jest następująca: „tablica to struktura, która przechowuje elementy tego samego typu”. Inną sprawa jest jej implementacja. W Javie zrobiono to w taki sposób, aby niemożliwe było zrobienie typowego błędu popełnianego przez początkujących, ale także zdarzającego się zaawansowanym programistom C/C++. Chodzi o pisanie i czytanie „za” zadeklarowaną tablicą. W Javie próba dostępu do nieistniejącego elementu tablicy kończy się po prostu błędem. W C/C++ skończyć się mogła różnie, ze spowodowaniem niestabilności systemu włącznie. W C/C++ tablica to po prostu fragment pamięci, z adresem zapamiętanym we wskaźniku do tablicy. W Javie tablice są obiektami, mogą więc kontrolować wykonywane na sobie operacje.

Pomimo że implementacja jest całkiem inna, praktyczne zasady dotyczące tablic w Javie są niemal identyczne jak w przypadku dynamicznie tworzonych tablic w C++. Możemy deklarować tablicę, co oznacza stworzenie referencji do tablicy, oraz ją zdefiniować, czyli zarezerwować pamięć, a następnie zapełnić ją zmiennymi lub obiektami.

typ`[]` *nazwa_referencji*`;` //deklaracja referencji do tablicy z elementami typu *typ*

nazwa_referencji`=new` *typ*`[`*wielkość*`];` //utworzenie tablicy operatorem `new`, zajęcie pamięci

Oto przykłady:

```
int[] ti=new int[100];
Button[] tb=new Button[100];
```

Jak widać, równie dobrze można tworzyć tablice typów prostych, jak i referencji do klas. Także klas napisanych przez użytkownika.

Tablice są automatycznie inicjowane zerami, a w przypadku referencji — referencjami `null`.

Uwaga! Utworzenie tablicy referencji nie oznacza, że powstały już jakiekolwiek obiekty.

W przypadku tablicy referencji zazwyczaj natychmiast tworzymy odpowiednie obiekty, np.

```
Button[] tb=new Button[100];
for (int i=0;i<tb.length;i++) tb[i]=new Button();
```

Po zakresie pętli `for`[7] widzimy, że elementy tablicy indeksowane są od 0. Zatem ostatni element ma indeks *wielkość*`-1`. W tym przykładzie indeksy tablicy to liczby od 0 do 99. Powyższy przykład pokazuje także, jak uzyskać dostęp do elementów tablicy. Korzystamy z nawiasów kwadratowych i numeru indeksu *tablica*`[`*indeks*`]`.

Możliwe jest również takie inicjowanie tablic:

```
int[] tj={1,2,3};
Button[] tc={new Button("1"),new Button("2"),new Button("3")};
```

W przypadku komponentów korzystanie z tablic daje korzyści. Można wiele operacji zamknąć w pętlach zamiast wypisywać je kolejno dla każdego obiektu. Np.

```
for (int i=0;i<tc.length;i++)
```

[7] Samą pętlę `for` bardziej szczegółowo omówię nieco niżej.

```
{
  this.setBounds(20,20+i*30,100,20);
  this.add(tc[i]);
}
```

Przykład projektowania w ten sposób interfejsu znajdzie się w rozdziale 4. skryptu (aplet `Puzzle`).

Podobnie jak w C/C++, aby zrealizować tablicę wielowymiarową, należy stworzyć tablicę tablic. Np.

```
int[][] tm0={{0,1,2},{3,4,5}};
```

lub

```
int[][] tm1=new int[2][3];
for(int i=0; i<tm1.length; i++)
  for(int j=0; j<tm1[i].length; j++)
    tm1[i][j]=3*i+j;
```

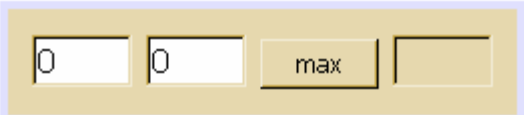
Po powyższych przykładach widać, że korzystanie z tablic jest zazwyczaj ściśle związane ze znajomością pętli `for`. Więcej szczegółów na ten temat znajdzie Czytelnik niżej w podrozdziale dotyczącym tej pętli.

Instrukcje sterujące

Opis instrukcji sterujących nie jest w zasadzie związany z żadnym językiem, choć oczywiście tutaj przykłady będą pisane w Javie. Zrozumienie instrukcji warunkowej, tworzenia pętli, a potem projektowania klas i operowania obiektami jest elementarzem współczesnego programisty, wiedzą podstawową, którą będzie mógł wykorzystać we wszystkich nowoczesnych językach, takich jak C++, Object Pascal, C# i nowe, które dopiero powstaną.

Ćwiczenie 2.1.

1. Tworzymy nowy projekt o nazwie *JezykJava*.
2. Dodajemy do niego aplet typu *Java GUI Forms*, *JApplet Form*. Nazwijmy go `JSterowanie`.
1. Na aplecie umieszczamy trzy pola edycyjne `JTextField` z palety komponentów *Swing* oraz przycisk `JButton`.
2. Domyślny tekst w polach edycyjnych ustawiamy za pomocą inspektora modyfikując właściwość `text` według wzoru na rysunku 2.1.
3. Po zmianie zawartości pól edycyjnych zmieni się ich szerokość. Zaznaczmy je wszystkie i ustawmy `Horizontal Size` na 50.
4. Etykietę przycisku zmieniamy modyfikując właściwość `text` zmieniając ją np. na `max`.
5. Trzecie pole tekstowe powinno mieć zablokowaną możliwość edycji (właściwość `editable` ustawiona na `false`).
6. Warto też zwiększyć rozmiar czcionek w polach edycyjnych do 16 (właściwość `font`).
7. Tworzymy metodę zdarzeniową dla przycisku związaną z jego kliknięciem (zdarzenie `actionPerformed`).



Rysunek 1. Interfejs apletu Sterowanie

<<koniec ćwiczenia>>

Po zakończeniu sekwencji czynności z ćwiczenia 2.1 na ekranie zobaczymy kod apletu `JSterowanie` z kursorem ustawionym wewnątrz stworzonej w ostatnim punkcie metody `jButton1ActionPerformed`. Wszystko jest gotowe do wpisywania przez nas kodu demonstrującego instrukcje sterujące.

Instrukcja warunkowa

Wyboru większej spośród dwóch wartości dokonujemy korzystając z operatora `>`, który zwraca wartość logiczną `true`, jeżeli to, co stoi przed nim, jest większe od tego, co stoi za nim. Nie ma w tym oczywiście żadnej niespodzianki. Inne dostępne operatory to `>=`, `<`, `<=`. Ich funkcje są całkowicie zgodne z tym, czego nauczyliśmy się w szkole podstawowej. Mniej oczywiste są operatory `==` (równy) i `!=` (różny). W pierwszym z nich wykorzystano podwójny znak równości, aby odróżnić od operatora przypisania[8]. W drugim wykrzyknik ma kojarzyć się z operatorem negacji wartości logicznej `!`.

Ćwiczenie 2.2.

Aby napisać metodę wybierającą większą z dwóch podanych przez użytkownika w polach edycyjnych liczb:

Do metody zdarzeniowej przygotowanej w poprzednim ćwiczeniu wstawiamy kod z listingu 2.1 (kod wewnątrz nawiasów klamrowych).

Listing 2.1. Metoda wybierająca większą spośród dwóch liczb

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        int a=Integer.valueOf(jTextField1.getText().trim()).intValue();
        int b=Integer.valueOf(jTextField2.getText().trim()).intValue();
        int c=a;
        if (b>a) c=b;
        jTextField3.setText(String.valueOf(c));
    }
```

<<koniec ćwiczenia>>

W pierwszej i drugiej linii definiowane są zmienne pomocnicze, które przechowują wartości wpisane przez użytkownika do pierwszego i drugiego pola tekstowego. Na wypadek, gdyby na początku lub końcu tekstu w polu tekstowym znalazły się niechciane spacje, przed konwersją stosowana jest metoda `String.trim`, która je usuwa[9]. Do zmiany łańcuchów z pól tekstowych na liczbę zastosowałem metodę `Integer.valueOf`. W ten sposób uzyskałem obiekt typu `Integer` — tj. obiekt poznanej wcześniej klasy opakowującej do liczb typu `int`. Aby zapisać obiekt klasy `Integer` do zmiennej typu `int`, użyłem metody konwertującej `Integer.intValue`.

Zapis w jednej linii może być nieco mylący. W rzeczywistości są bowiem wykonywane aż cztery metody jedna po drugiej. Przepisując polecenie z pierwszej linii metody, w taki sposób, żeby w każdej linii znalazło się wywołanie tylko jednej metody, otrzymamy:

```
String aStr=textField1.getText();
aStr=aStr.trim();
Integer aInt=Integer.valueOf(aStr);
int a=aInt.intValue();
```

W dwóch zaznaczonych liniach wykonywana jest zasadnicza część metody, a mianowicie wybór spośród wartości zmiennych `a` i `b` tej, która ma większą wartość, i zapisanie jej do `c`. Najpierw deklarowana i inicjowana jest zmienna `c` wartością z `a`. Następnie korzystamy z konstrukcji

```
if (warunek) polecenie;
```

która wykonuje *`polecenie`* pod warunkiem, że *`warunek`* ma wartość `true`. W naszym przypadku zmiana wartości `c` odbędzie się pod warunkiem, że wartość `b` jest większa od `a`.

Identyczne działanie miałaby następująca konstrukcja

```
int c;
```

---

[8] A propos. Należy pamiętać, że w Javie, podobnie jak w C++, pojedynczy znak `=` to operator przypisania, a podwójny `==` porównania. Nie jest to wiedza, którą trzeba mieć cały czas na uwadze, bo w Javie, w przeciwieństwie do jej starszego brata C++, istnieje wyraźne rozróżnienie między warunkiem zwracającym wartość logiczną a poleceniem. Nie jest dopuszczalna konstrukcja typu `if (a=0) b=0;`, która jest dopuszczalna w C++. Warunek w `if` musi mieć wartość logiczną, a w Javie nie ma jej operator przypisania.

[9] W ten sposób chronimy się przed błędem, który zgłoszony zostałby podczas konwersji w przypadku obecności spacji.

```
if (b>a) c=b; else c=a;
```

Tym razem wykorzystaliśmy pełną postać instrukcji warunkowej `if`:

```
if (warunek) polecenie_jeżeli_prawda; else polecenie_jeżeli_fałsz;
```

Polecenie następujące za `else` będzie wykonane wówczas, gdy *`warunek`* nie jest spełniony, a więc mówiąc językiem Javy, gdy ma wartość `false`.

W takim przypadku jak powyższy, gdy od jakiegoś warunku zależy wartość zapisywana do zmiennej, warto korzystać z odziedziczonego z C/C++ operatora warunku:

```
int c=(b>a)?b:a;
```

Nawiasy dodane zostały dla poprawienia czytelności. Jakie jest działanie tego operatora? Zwraca ona wartość w zależną od sprawdzanego warunku:

```
warunek?wartość_jeżeli_prawda:wartość_jeżeli_fałsz
```

Ostatnie polecenie metody `jButton1ActionPerformed` zmieniające liczbę `c` w łańcuch, który jest umieszczany w polu tekstowym, można nieco uprościć przepisując je w znany już nam sposób korzystający z niejawnej konwersji przy operatorze `+` (zobacz podrozdział dotyczący łańcuchów):

```
jTextField3.setText(""+c);
```

Z takiej postaci będę zazwyczaj korzystał w przypadku konieczności konwersji liczb na łańcuch w dalszych rozdziałach.

Po wszystkich zmianach metoda zdarzeniowa może przybrać postać zaprezentowaną na listingu 2.2 (dodatkowa modyfikacja to wyeliminowanie zmiennej `c`):

Listing 2.2. Zmodyfikowana metoda z ćwiczenia 2.2

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    int a=Integer.valueOf(textField1.getText().trim()).intValue();
    int b=Integer.valueOf(textField2.getText().trim()).intValue();
    jTextField3.setText(""+(b>a?b:a));
  }
```

Nawiasy wokół operatora warunkowego w ostatniej linii zostały użyte, aby zachować prawidłową kolejność działań.

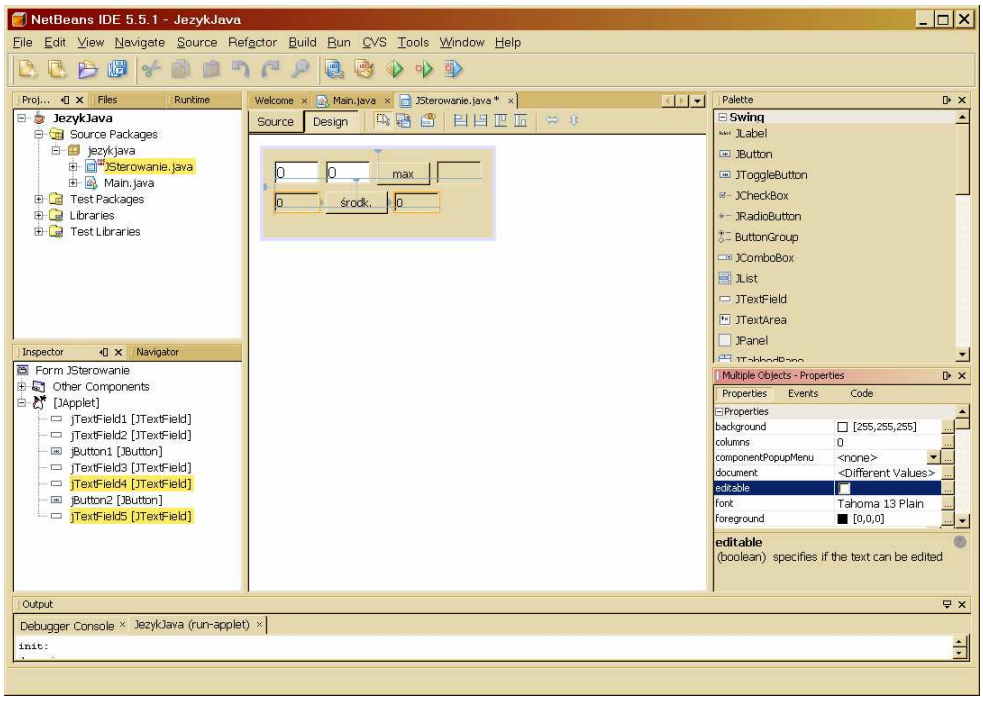
Złożone warunki i operatory logiczne

Warunek może być oczywiście bardziej skomplikowany niż proste wykorzystanie operatorów porównania `<`, `>` lub `==`. Spójrzmy na poniższy przykład, w którym aplet `JSterowanie` uzupełnimy o kolejny przycisk i dwa pole tekstowe. Po naciśnięciu przycisku do pierwszego z dodanych pól tekstowych ma być zapisywana losowa liczba całkowita z zakresu –10 do 10, a do drugiego wartość „środkowa” spośród trzech wartości zapisanych w `textField1` i `textField2` oraz losowo wybranej przed chwilą wartości.

Ćwiczenie 2.3.

Aby rozbudować interfejs apletu `JSterowanie` przez dodanie dwóch pól edycyjnych i przycisku:

1. Postępując podobnie jak w poprzednim ćwiczeniu, dodajemy do interfejsu apletu dodatkowe komponenty z palety *Swing*: przycisk `JButton` i dwa pola edycyjne `JTextField` według pokazanego na rysunku 2 wzoru.
2. Modyfikujemy tekst w polach edycyjnych oraz etykietę przycisku ustawiając ją na *środk.*.



Rysunek 2. Uzupełniony interfejs apletu Sterowanie

3. Następnie tworzymy domyślną metodę zdarzeniową do nowego przycisku klikając go dwukrotnie w podglądzie na zakładce *Design*.

4. Do powstałej metody `jButton2ActionPerformed` wpisujemy instrukcje definiujące zmienne `a` i `b` zainicjowane wartościami odczytanymi z pól edycyjnych, jak w poprzednim ćwiczeniu, a ponadto definicję zmiennej `c`, która inicjowana jest wartością losową z przedziału od –10 do 10, oraz instrukcję pokazującą wartość tej zmiennej w polu edycyjnym:

```
int a=Integer.valueOf(jTextField1.getText().trim()).intValue();
int b=Integer.valueOf(jTextField2.getText().trim()).intValue();
int c=(int)(20*Math.random()-10);
jTextField4.setText(""+c);
```

<<koniec ćwiczenia>>

Teraz zaczyna się zabawa. Problem jest prosty: jak w najbardziej optymalny sposób sprawdzić, która spośród trzech zmiennych nie jest ani największa, ani najmniejsza?

Można oczywiście brać po kolei każdą liczbę i sprawdzać, czy wśród pozostałych jest jednocześnie liczba o mniejszej i większej wartości. Liczba porównań jest w takim wypadku oczywiście duża:

```
int d=0;
if ((a<b && a>c) || (a>b && a<c)) d=a;
if ((b<a && b>c) || (b>a && b<c)) d=b;
if ((c<a && c>b) || (c>a && c<b)) d=c;
```

W instrukcjach warunkowych wykorzystaliśmy operatory logiczne `&&` i `||`. Nie wolno ich pomylić z operatorami bitowymi `&` i `|`[10]. Operator `&&` implementuje koniunkcję zwykle zapisywaną jako `AND`. Operator ten zwraca prawdę, jeżeli oba argumenty są prawdziwe. Drugi operator `||` (`OR`, alternatywa) zwraca prawdę, gdy przynajmniej jeden z jego argumentów jest prawdziwy.

[10] Sytuacja jest bardziej złożona, bo operatory `&` i `|` są przeciążone. Funkcjonują jako operatory binarne, jeżeli ich argumentami są liczby, zwracają liczbę z wykonaną operacją na bitach. Jeżeli argumentami są wartości logiczne, wówczas działają tak jak operatory `&&` i `||`. Różnica polega na tym, że `&&` i `||` są zoptymalizowane w taki sposób, że nie jest sprawdzana wartość drugiego argumentu, jeżeli z pierwszego jednoznacznie wynika, jaka powinna być wartość zwracana przez operator. Może to być niewygodne, gdy argumentem jest metoda, którą chcemy „przy okazji” uruchomić. Wówczas należy skorzystać z operatorów `&` i `|`.

Powyższy kod możemy od razu nieco zoptymalizować usuwając ostatnią linię i zmieniając inicjację zmiennej `d`. Pomysł polega na tym, że jeżeli nie są spełnione warunki z pierwszej i drugiej linii warunków, to jego spełnienie w trzeciej linii jest konieczne:

```
int d=c;
if ((a<b && a>c) || (a>b && a<c)) d=a;
if ((b<a && b>c) || (b>a && b<c)) d=b;
```

Wówczas z 12 porównań zostaje 8.

Kolejne optymalizacje opierają się na zapamiętaniu wyników pośrednich porównań (listing 2.3).

Listing 2.3 Metoda zdarzeniowa wybierająca wartość nie najmniejszą i nie największą spośród trzech liczb

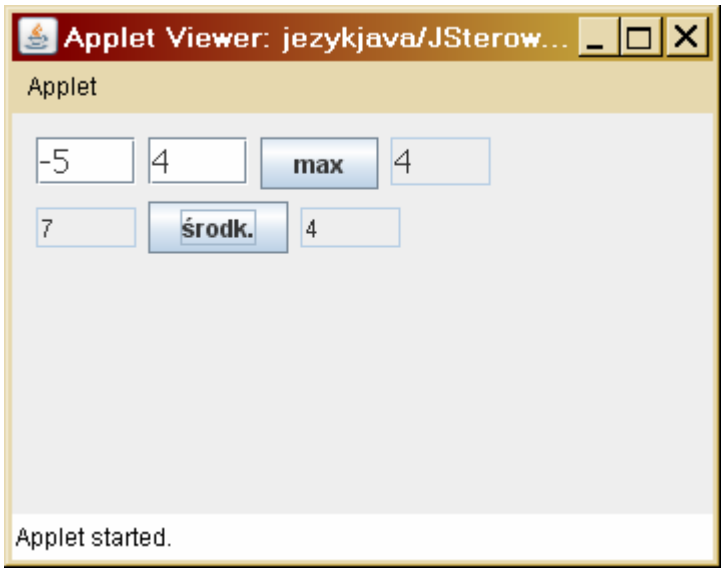
```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {

    int a=Integer.valueOf(jTextField1.getText().trim()).intValue();
    int b=Integer.valueOf(jTextField2.getText().trim()).intValue();
    int c=(int)(20*Math.random()-10);
    jTextField4.setText(""+c);

    int d=c;
    int t1=(a<b)?a:b; if (c<t1) d=t1;
    int t2=(a>b)?a:b; if (c>t2) d=t2;

    jTextField5.setText(""+d);
  }
```

Najpierw szukamy liczby o mniejszej wartości spośród `a` i `b`. Tę wartość zapisujemy do `t1`. Jeżeli `t1` jest większa od `c`, to `t1` jest na pewno liczbą środkową. Następnie szukamy większej w parze `a` i `b`. Jeżeli jest ona mniejsza od `c`, to jest również liczbą środkową. Pozostaje ostatnia możliwość, gdy `c` jest większa od `t1 = min(a,b)` i mniejsza od `t2 =max(a,b)`, ale on jest spełniony automatycznie, gdy dwa wcześniejsze okażą się fałszywe. Pozostają zatem tylko 4 porównania. I nie mam — szczerze mówiąc — już pomysłu, jak można by ich ilość jeszcze zmniejszyć.



Rysunek 3. Aplet JSterowanie uruchomiony w Applet Viewer

Ćwiczenie 2.4.

Aby w aplecie `Sterowanie` w polu tekstowym `jTextField5` umieścić wartość liczby „środkowej”:

Metodę `button2_actionPerformed`, której pisanie rozpoczęliśmy w poprzednim ćwiczeniu, uzupełniamy o wytłuszczony fragment z listingu 2.3.

<<koniec ćwiczenia>>

Poza omówionymi dwuargumentowymi operatorami logicznymi Java dostarcza także jednoargumentowy operator negacji `!` zamieniający wartość logiczną argumentu oraz rzadziej używany[11] operator `^` alternatywy wykluczającej `XOR` prawdziwy tylko, gdy jeden z argumentów jest prawdziwy.

Implikacja. Definiowanie metod

Za pomocą dostępnych operatorów logicznych, bez korzystania z instrukcji warunkowej `if`, zdefiniujemy metodę implementującą logiczną operację implikacji, tj. zwracającą fałsz tylko wtedy, gdy pierwszy argument jest prawdziwy, a drugi fałszywy.

Operacja implikacji ma opisywać poprawność wnioskowania. W przeciwieństwie do wszystkich opisywanych wyżej operacji dwuargumentowych implikacja nie jest symetryczna. Pierwszy argument określa prawdziwość przesłanek, a drugi wniosków. Implikacja jest fałszywa tylko wówczas, gdy z prawdziwych wniosków uzyskaliśmy fałszywy wniosek.

Ćwiczenie 2.5.

Aby za pomocą dostępnych operatorów logicznych zdefiniować metodę implementującą implikację:

Przygotujmy metodę `implikacja` w naszym aplecie `JSterowanie` zgodnie z listingiem 2.4 (wpisujemy oczywiście tylko wytłuszczoną część, pozostałe fragmenty kodu przedstawiłem, aby ułatwić znalezienie właściwego miejsca).

Listing 2.4. Definiując metodę, należy uważnie wybrać miejsce, w którym wpisujemy jej kod – musi znaleźć się wewnątrz klasy JSterowanie

```
static boolean implikacja(boolean p, boolean w)
{
   return !(p && !w);
}
```

<<koniec ćwiczenia>>

Metoda musi być zdefiniowana w obrębie klasy, ale nie w innej metodzie. Najłatwiej będzie, gdy uważnie znajdziemy ostatni nawias klamrowy klasy `JSterowanie` (za nią są następne klasy, które odpowiadają za wywoływanie metod zdarzeniowych) i przed nim wstawimy nową metodę.

Wartość argumentu `p` w metodzie `impikacja` z listingu 2.4 to prawdziwość przesłanki, natomiast `w` to prawdziwość wniosku. Zgodnie z definicją implikacji przedstawioną powyżej wartość fałszywa będzie zwracana tylko wówczas, gdy przesłanka jest prawdziwa, a wniosek fałszywy, a zatem wartość implikacji „jeżeli p to w” równoważna jest stwierdzeniu „nieprawda, że przesłanka jest prawdziwa, a wniosek fałszywy”. Spójnik *a* pełni w tym zdaniu tak naprawdę rolę logicznego *i*. A zatem implikacja równoważna jest twierdzeniu „nieprawda, że (p i nie w)” czyli w języku logiki *~ (p ^ !w)*, a w języku Javy `!(p && !w)`.

Instrukcja wielokrotnego wyboru

Instrukcję `switch` wykorzystamy do wyboru jednej z operacji logicznych.

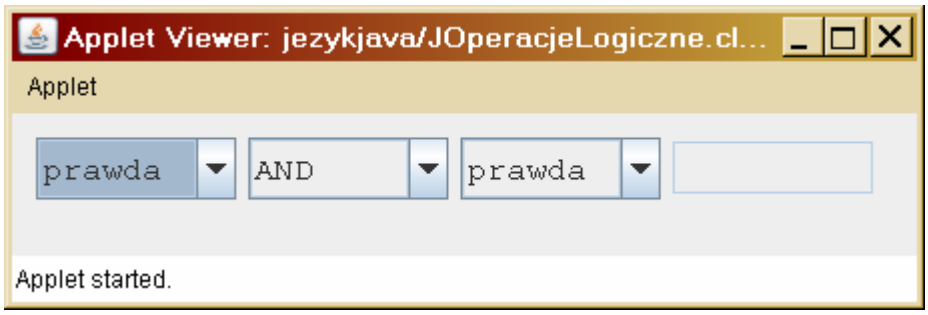
Ćwiczenie 2.6.

Aby przygotować interfejs apletu obliczającego wartość dwuargumentowych operatorów logicznych:

1. Za pomocą kreatora tworzymy nowy aplet o nazwie `JOperacjeLogiczne` w pakiecie `jezykjava` (zobacz ćwiczenie 1.9).

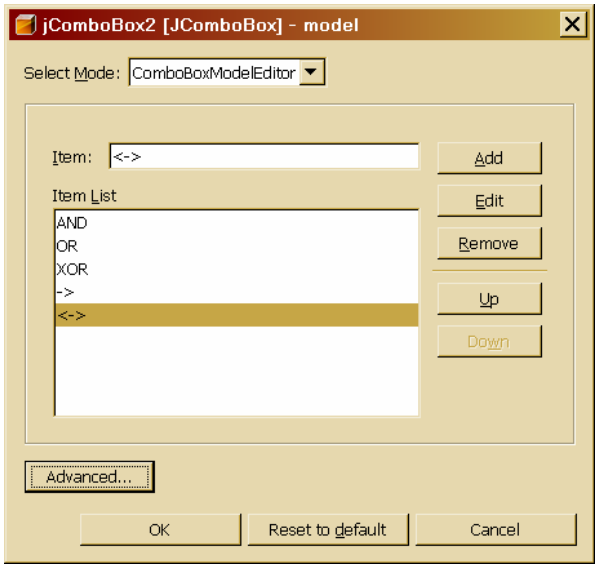
[11] Jest on rzadko używany głównie ze względu na brak prostego intuicyjnego odniesienia do języka potocznego, jakie mamy w przypadku `AND` i `OR`. Operator ten pełni w zasadzie rolę słowa „albo”, ale jego intuicja zaciera się, bo słowo „albo” to może być używane również w roli „lub”.

2. Przechodzimy do widoku projektowania.
3. Umieszczamy na aplecie trzy rozwijane listy `JComboBox` oraz jedno pole tekstowe `JTextField` (rysunek 2.3).
4. Zmieniamy czcionkę we wszystkich komponentach (można to zrobić, jeżeli zaznaczymy wszystkie komponenty na podglądzie lub w oknie struktury apletu) na *Monospaced* o rozmiarze 16.



Rysunek 4. Interfejs apletu JOperacjeLogiczne

5. Za pomocą edytora własności związanego z własnością model rozwijanych list (rysunek 5) zmieniamy zawartość list na
   a) jComboBox1, jComboBox3: „prawda”, „fałsz”.
   b) jComboBox2: „AND”, „OR”, „XOR”, „->”, „<->”



Rysunek 5. Edytor własności model

<<koniec ćwiczenia>>

W rozwijanych listach `jComboBox1` i `jComboBox3` znajdą się tylko po dwie pozycje „prawda” i „fałsz”. Lista `jComboBox2` zawiera, jak widać w powyższym kodzie, operatory logiczne włącznie ze zdefiniowaną przez nas implikacją i równoważnością.

Ćwiczenie 2.7.

Aby zdefiniować silnik apletu `OperacjeLogiczne`:

1. Następnie tworzymy metodę zdarzeniową dla dla zdarzenia `itemStateChanged` pierwszej rozwijanej listy.
2. W metodzie tworzymy zmienne pomocnicze `l` i `r`, zmienną `w`, do której zapisany będzie wynik, oraz pomocniczą zmienną, którą wykorzystamy w razie wystąpienia błędu:

```
private void jComboBox1ItemStateChanged(java.awt.event.ItemEvent evt) {
        boolean l=(jComboBox1.getSelectedItem()=="prawda")?true:false;
```

```
        boolean r=(jComboBox3.getSelectedItem()=="prawda")?true:false;
        boolean w;
        boolean error=false;
    }
```

3. Następnie zastosujemy instrukcję wyboru do obliczenia wartości logicznej będącej wynikiem właściwej, ze względu na wybór w komponencie `jComboBox2`, operacji logicznej. Argumentem instrukcji `switch` musi być liczba całkowita (`byte`, `short`, `int`, ale nie `long`) lub znak (`char`). W naszym przypadku argumentem instrukcji `switch` będzie indeks wybranej pozycji (numerowany od 0) zwracany przez metodę `jComboBox2.getSelectedIndex`. Cała instrukcja `switch` powinna mieć zatem następującą postać:

```
switch(jComboBox2.getSelectedIndex())
{
   case 0: w=l && r; break; //AND
   case 1: w=l || r; break; //OR
   case 2: w=l ^ r; break;  //XOR
   case 3: w=JSterowanie.implikacja(l,r); break; //->
   case 4: w=(l==r); break; //<->
   default: w=false; error=true;
}
```

4. Na końcu metody umieszczamy także polecenie wyświetlające wynik w polu tekstowym:

```
jTextField1.setText(""+w);
```

5. Tak przygotowaną metodę związmy z pozostałymi dwoma rozwijanymi listami podając w inspektorze przy zdarzeniu `itemStateChanged` nazwy metody `jComboBox1ItemStateChanged`.

6. Aby po uruchomieniu aplet wyglądał przyzwoicie, powinniśmy uruchomić tę metodę także zaraz po uruchomieniu apletu. W tym celu w metodzie `init`, po poleceniach zapełniających rozwijane listy, należy również dodać uruchomienie metody `jComboBox1ItemStateChanged` z argumentem `null` (argumentu nie wykorzystywaliśmy w ciele metody, więc jego wartość nie ma znaczenia).

<<koniec ćwiczenia>>

Ostatnia operacja, określona w zawartości `choice2` jako `<->`, to operacja równoważności. Jest ona prawdziwa dla `l` i `r`, gdy `l -> r` i jednocześnie `r -> l`, a więc tylko wówczas, gdy prawdziwa jest implikacja w obie strony. Okazuje się, że to jest prawdą wtedy, gdy oba argumenty mają równe wartości. Zatem operacja równoważności jest zaimplementowana w dobrze nam znanym operatorze porównania `==`, jeżeli użyjemy go dla argumentu typu `boolean`.

Pętla for

Pętla `for`, której składnia jest następująca:

```
for(inicjacja; warunek; modyfikacja_indeksu) polecenie;
```

wykonuje *`polecenie`* tak długo, dopóki spełniony jest *`warunek`*. Polecenie *`inicjacja`* wykonywane jest tylko raz, polecenie *`modyfikacja_indeksu`* w każdej iteracji pętli i daje możliwość zmiany wartości indeksu. Warunek, który sprawdzany jest przed każdą iteracją, musi zwracać wartość logiczną.

Jeżeli zainicjujemy indeks pętli `i` wartością 0, to aby wykonać 100 iteracji, musimy ustalić warunek równy `i`<100. Wówczas maksymalną wartością `i` będzie 99. W naszej pętli będzie to wyglądać następująco:

```
for (int i=0;  i<100; i++)
```

Pętle `for` można skonstruować w zupełnie dowolny sposób. Może w ogóle nie mieć indeksu, można indeks zmieniać w dowolny sposób, nie tylko o jeden, ale z dowolnym skokiem „w górę” lub „w dół”. Zobacz kilka przykładów (pomijam w nich polecenie wykonywane w każdym kroku):

```
for(int i=99; i>=0; i--)
for(double i=0; i<=10.0; i=i+0.1)
for(int i=2; i<=256; i=i*i)
```

Można też w ogóle zdegenerować pętle. Oto przykład

```
int i=0;
for(;;)
  {
    if ((i++)>=100) break;
  }
```

Jednak jedyny praktyczny pożytek z tego przykładu to poznanie działania polecenia `break`, które przerywa ciąg poleceń umieszczonych w nawiasach klamrowych i kończy pętle.

Przy bardziej wymyślnych konfiguracjach należy być ostrożnym, bo łatwo o zapętlenie, tj. stworzenie pętli wykonywanej w nieskończoność.

Niemal we wszystkich przykładach w tej książce będziemy korzystać z pętli `for` w jej podstawowej postaci z pierwszego przykładu, a więc korzystającej z lokalnie zdefiniowanego indeksu zainicjowanego przez 0, z warunkiem postaci *`indeks<ilość_iteracji`* oraz zwiększaniem indeksu o jeden w każdym kroku operatorem inkrementacji *`indeks++`*. W ten sposób obsługa indeksu jest całkowicie zamknięta w nawiasie przy poleceniu `for`.

Aby zademonstrować pętlę `for` w akcji, zaprojektujemy aplet, który symuluje następujący eksperyment z dwoma kostkami do gry. Rzucamy wiele razy, powiedzmy 100, dwoma kostkami do gry i sumujemy liczbę oczek na obu kostkach uzyskując za każdym razem wynik z zakresu od 2 do 12. Powtarzając rzut dwoma kostkami wiele razy, zliczamy, ile razy otrzymujemy poszczególne wyniki z tego przedziału. Wynik pokażemy na 11 suwakach (paskach przewijania) z biblioteki *AWT*.

Ćwiczenie 2.8.

Aby napisać aplet symulujący wyniki uzyskane przy rzucaniu dwoma kostkami:

1. Za pomocą kreatora tworzymy nowy aplet o nazwie `JDwieKostki` w pakiecie `JezykJava`.
2. Przechodzimy do widoku projektowania (zakładka *Design*).
3. Umieśćmy na aplecie 11 suwaków `JScrollbar` (możemy położyć jeden, a reszte stworzyć przez kopiowanie *Ctrl+C*, *Ctrl+V*). Ich właściwość `orientation` jest domyślnie ustawiona na `VERTICAL`, co powoduje, że paski przewijania są ustawione pionowo. I bardzo dobrze.
4. Do apletu dodajemy również jeden przycisk z etykietą `Rzucaj kostkami`.
5. Stwórzmy metodę zdarzeniową dla tego zdarzenia `actionPerformed` przycisku klikając go dwukrotnie na podglądzie apletu.

   a) Wewnątrz metody deklarujemy zmienną `n` określającą ilość rzutów oraz tablicę `wyniki` przechowującą dla każdej sumy oczek liczbę uzyskanych trafień:

   ```
   private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
       int n=100;
       int[] wyniki=new int[13];
     }
   ```

   b) Następnie tworzymy pętlę `for`, którą wstawiamy do metody utworzonej w poprzednim podpunkcie:

   ```
   for(int i=0;i<n;i++)
   {
      byte pierwszaKostka=(byte)Math.floor(6*Math.random()+1);
      byte drugaKostka=(byte)Math.floor(6*Math.random()+1);
      wyniki[pierwszaKostka+drugaKostka]++;
   }
   ```

   c) Teraz serią metod `JScrollBar.setValue` ustalamy pozycję każdego suwaka. Trzeba pamiętać, że dla wartości 0 pozycja suwaka jest u góry. Musimy zatem ustalić jego właściwość `value` równą położeniu maksymalnemu minus właściwa wartość ilości trafień.

   ```
   int max=jScrollBar1.getMaximum();
   ```

```
jScrollBar1.setValue(max-wyniki[2]);
jScrollBar2.setValue(max-wyniki[3]);
jScrollBar3.setValue(max-wyniki[4]);
jScrollBar4.setValue(max-wyniki[5]);
jScrollBar5.setValue(max-wyniki[6]);
jScrollBar6.setValue(max-wyniki[7]);
jScrollBar7.setValue(max-wyniki[8]);
jScrollBar8.setValue(max-wyniki[9]);
jScrollBar9.setValue(max-wyniki[10]);
jScrollBar10.setValue(max-wyniki[11]);
jScrollBar11.setValue(max-wyniki[12]);
```

6. Tak przygotowany aplet kompilujemy i uruchamiamy (klawisz *Shift+F6*).

<<koniec ćwiczeniu>>

Tablica `wyniki` zadeklarowana w punkcie 8a jest o 2 pozycje za duża. Mamy tylko 11 możliwości od najmniejszej sumy 1 + 1 = 2 do największej 6 + 6 = 12. Zrobiłem tak jednak, jak zresztą robię zazwyczaj w takich sytuacjach, aby liczba oczek w kombinacji odpowiadała numerowi indeksu w tablicy. Dla przejrzystości kodu warto stracić dwa bajty.

Przy każdej iteracji pętli napisanej w punkcie 8b do zmiennych `pierwszaKostka` i `drugaKostka` zapisywane są liczby oczek z poszczególnych rzutów. Następnie są sumowane, a element tablicy `wyniki` odpowiadający uzyskanej sumie jest zwiększany o jeden. W ten sposób w tablicy `wyniki` znajdują się zliczenia trafień w poszczególne liczby oczek.

Po uruchomieniu apletu i naciśnięciu przycisku *Rzucaj kostkami* okazuje się, że zmiana położenia suwaków jest ledwo widoczna. Konieczne jest przeskalowanie właściwości zakresu wartości pokazywanych przez suwaki. A mówiąc wprost, należy sprawdzić, jaka jest maksymalna ilość trafień w tabeli `wyniki` i dopasować do niej własność `maximum` każdego suwaka. Zrobimy to znowu korzystając z pętli `for` przebiegającej po wszystkich używanych elementach tabeli `wyniki` i szukającej elementu największego. Idea jest prosta. Ustalamy wartość zmiennej `max` równą ilości trafień dla sumy dwóch oczek, a następnie sprawdzamy po kolei, czy liczba trafień dla pozostałych oczek nie jest większa niż zapisana w `max`. Jeżeli jest — aktualizujemy wartość `max` i szukamy dalej aż do końca tablicy `wyniki`.

Ćwiczenie 2.9.

Aby za pomocą pętli `for` ustalić maksymalną ilość trafień pośród wszystkich przedziałów:

Do metody zdarzeniowej `jButton1ActionPerformed` przed serią wywołań metod `JScrollBar.setValue` dopisujemy kod, który powinien zastępować dotychczasowe ustalanie wartości zmiennej `max`:

```
int max=wyniki[2];
for(int i=3; i<=12; i++)
{
  if(wyniki[i]>max) max=wyniki[i];
}
```

<<koniec ćwiczenia>>

Zaraz po ustaleniu właściwej wartości `max` powinniśmy zmienić właściwość `maximum` każdego suwaka. Moglibyśmy to zrobić serią wywołań metody `setMaximum`, podobnie jak dla `setValue`, ale skorzystamy z pewnej sztuczki i rzecz wykonamy w pętli. Ponownie będzie to pętla typu `for`.

Ćwiczenie 2.10.

Aby za pomocą pętli `for` przeskalować zakres suwaków tak, aby najlepiej zobrazować uzyskane wyniki:

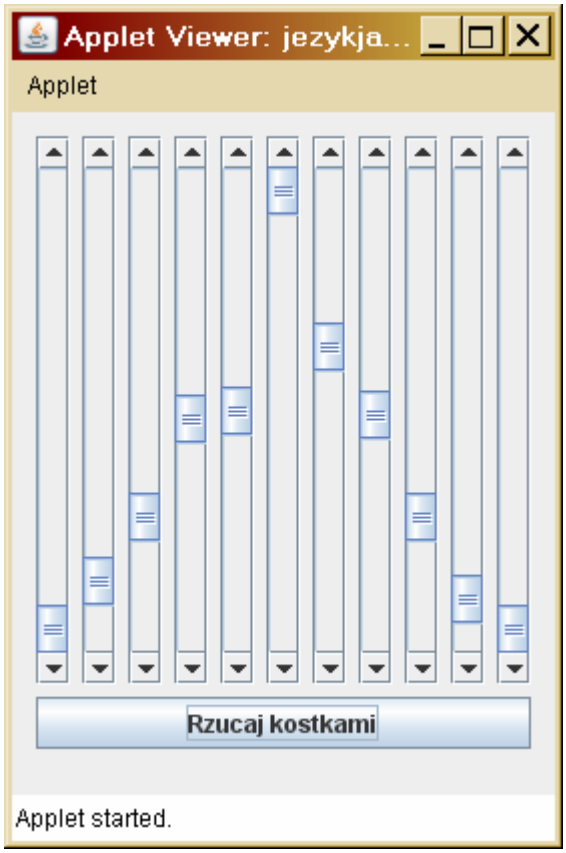
Dodajemy: `import javax.swing.JScrollBar;`.

Do metody `jButton1ActionPerformed`, za kodem wpisanym w poprzednim ćwiczeniu, dodajemy pętlę `for`:

```
      for(int i=0; i<this.getComponentCount(); i++)
      {
         java.awt.Component komponent = this.getComponent(i);
         if (komponent instanceof JScrollBar)
         {
            JScrollBar suwak = (JScrollBar) komponent;
            suwak.setMaximum(max);
            suwak.setVisibleAmount(max/20);
         }
      }
```

<<koniec ćwiczenia>>

Wykonujemy pętle z indeksem `i` od 0 do wartości odczytanej za pomocą metody apletu `this.getComponentCount`. Metoda ta zwraca ilość komponentów, które zostały wcześniej dodane do apletu poleceniem `this.add`, a więc ilość komponentów umieszczonych na jego powierzchni. Następnie po kolei czytamy referencję każdego komponentu i sprawdzamy za pomocą operatora `instanceof`, czy jest typu `JScrollbar` (do skonstruowania tego warunku można wykorzystać szablon *Ctrl+J*, inst). Jeżeli komponent jest obiektem typu `JScrollbar`, to rzutujemy referencję pobraną metodą `this.getComponent` na referencję do `JScrollbar`, co da nam dostęp do wszystkich metod tego komponentu. Następnie za pomocą metody `JScrollbar.setMaximum` ustawiamy jego właściwość `maximum`. Dodatkowo zmieniamy także właściwość `visibleAmount`, która decyduje o wielkości przesuwanego elementu w suwaku. Cała ta pętla powinna znaleźć się po ustaleniu wartości zmiennej `max`, ale przed ustaleniem położenia suwaków.



Rysunek 6. Symulacja rzutów dwoma kostkami

Po tych zmianach możemy obejrzeć wreszcie rozkład liczby oczek. Po każdym naciśnięciu przycisku z etykietą `Rzucaj kostkami` oglądamy nowy przypadek.

Proszę zauważyć, że napisaliśmy naszą metodę w taki sposób, że zmiana ilości rzutów kostką to żaden problem. Wystarczy zmienić inicjację zmiennej `n`. Jednak zamiast tego proponuję zrobić coś innego: przenieśmy deklarację tabeli `wyniki` przed metodę. Wówczas po każdym naciśnięciu przycisku tabela nie będzie tworzona

od nowa, a zliczenia rzutów będą się kumulować. Zobaczymy, jak w miarę zwiększania łącznej ilości rzutów rozkład częstości się wygładza. Listing 2.6 pokazuje całą metodę przygotowaną w ćwiczeniach 2.8 – 2.10.

Listing 2.6 Metoda implementująca doświadczenie z dwoma kostkami

```
int[] wyniki=new int[13]; //2 pierwsze elementy nie będą nigdy wykorzystane
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    int n=100;
    //int[] wyniki=new int[13];

    for(int i=0;i<n;i++)
    {
        byte pierwszaKostka=(byte)Math.floor(6*Math.random()+1);
        byte drugaKostka=(byte)Math.floor(6*Math.random()+1);
        wyniki[pierwszaKostka+drugaKostka]++;
    }

    //int max=jScrollBar1.getMaximum();
    int max=wyniki[2];
    for(int i=3; i<=12; i++)
    {
        if(wyniki[i]>max) max=wyniki[i];
    }
    for(int i=0; i<this.getComponentCount(); i++)
    {
        java.awt.Component komponent = this.getComponent(i);
        if (komponent instanceof JScrollBar)
        {
           JScrollBar suwak = (JScrollBar) komponent;
           suwak.setMaximum(max);
           suwak.setVisibleAmount(max/20);
        }
    }
    jScrollBar1.setValue(max-wyniki[2]);
    jScrollBar2.setValue(max-wyniki[3]);
    jScrollBar3.setValue(max-wyniki[4]);
    jScrollBar4.setValue(max-wyniki[5]);
    jScrollBar5.setValue(max-wyniki[6]);
    jScrollBar6.setValue(max-wyniki[7]);
    jScrollBar7.setValue(max-wyniki[8]);
    jScrollBar8.setValue(max-wyniki[9]);
    jScrollBar9.setValue(max-wyniki[10]);
    jScrollBar10.setValue(max-wyniki[11]);
    jScrollBar11.setValue(max-wyniki[12]);
}
```

Pętle while i do… while

W Javie dostępne są również pętle postaci

```
while(warunek) {polecenia}
```

oraz

```
do {polecenia} while(warunek);
```

Różnica między nimi polega na tym, że w `while` *`warunek`* sprawdzany jest przed wykonaniem *`polecenia`*, a w `do...while` po ich wykonaniu. Z tego wynika, że *`polecenia`* w drugim rodzaju pętli zostaną wykonane przynajmniej raz, nawet jeżeli warunek nie jest nigdy spełniony.

Oto przykłady pętli ze 100 iteracjami od 0 do 99 (jeżeli wartość indeksów jest sprawdzana przed zwiększaniem o jeden za pomocą operatora ++). Po zakończeniu indeks ma w obu przypadkach wartość 100:

```
//pętla while
int i=0;
while(i<100) {i++;}


//pętla do..while
int j=0;
do {j++;} while(j<100);
```

A to przykład pokazujący różnicę w działaniu obu pętli:

```
while(false){System.out.println("while");} //tu pojawi się błąd kompilatora
do{System.out.println("do..while");}while(false);
```

Polecenie z pętli `do...while` zostanie raz wykonane, choć warunek jest zawsze fałszywy. Polecenie z pętli `while` nie zostanie wykonane ani razu, co więcej — pętla `while` w tej postaci nie zostanie nawet przepuszczona przez kompilator, który znajdzie fragment kodu, który nigdy nie zostanie wykonany i zgłosi to jako błąd.

Wykorzystane w powyższym przykładzie polecenie `System.out.println` wysyła łańcuch będący argumentem do standardowego strumienia wyjścia.

Klasy i obiekty

Alternatywny opis: *http://www.fizyka.umk.pl/~jacek/dydaktyka/klasy.html*, wybór języka: Java

Nie warto unikać klas i obiektów!

Programiści zazwyczaj długo unikają „przesiadki” z programowania strukturalnego na programowanie zorientowane obiektowo. Programujący w C++ mają możliwość zwlekania zdefiniowania pierwszej klasy w nieskończoność bojąc się wszystkich związanych z tym trudności i nowości. Ale gdy zacznie się już na poważnie programować obiektowo, po przedarciu się przez początkowe problemy (nie za bardzo da się zacząć programować obiektowo po trochu) szybko docenia się to, ilu błędów można uniknąć dzięki takiemu podejściu. Atomizacja kodu, oddzielenie interfejsu od ukrytej implementacji, dziedziczenie pozwalające na unikanie modyfikowania dobrze działających fragmentów kodu, a jedynie na jego rozwijanie, a nade wszystko możliwość stworzenia struktury klas odpowiadającej strukturze opisywanej przez program sytuacji, co ułatwia myślenie o tym co, a nie jak mam zaprogramować — wszystkie te udogodnienia przemawiają za programowaniem obiektowym.

Pojawiają się oczywiście nowe zagrożenia w programowaniu obiektowym w C++ — najważniejszym z nich jest wyciek pamięci przy dynamicznym tworzeniu obiektów i zagrożenia płynące z korzystania ze wskaźników. Nie są to oczywiście zagrożenia związane wyłącznie z obiektami, ale przy programowaniu obiektowym ujawniają się w sposób szczególnie częsty.

Jeżeli programujemy aplety i aplikacje w Javie za pomocą samego edytora, jesteśmy zmuszeni do „ręcznego” tworzenia klas i obiektów. Klasą musi być sam aplet i aplikacja, korzystanie z mechanizmu zdarzeń także zmusza do rozszerzania klas. NetBeans odsuwa od nas tę konieczność, ale nie na długo. Jeżeli zechcemy stworzyć wątek lub rozszerzyć funkcjonalność okna, a to tylko dwa drobne przykłady, zmuszeni będziemy stworzyć samodzielnie klasę. Warto więc nauczyć się programowania obiektowego, nawet jeżeli jeszcze nie stoimy przed bezwzględną koniecznością. Wcale nie ma konieczności poprzedzania nauki programowania obiektowego nauką jego starszej siostry — programowania strukturalnego. Co więcej, stwarza to zagrożenie, że programować obiektowo nigdy nie zaczniemy, gdyż często zdarza się, że młody programista mówi sobie „mogę wszystko zrobić korzystając z funkcji i zmiennych, po co mam brnąć w definiowanie klas?”. I ma oczywiście rację mówiąc, że definiowanie klas nie zwiększa możliwości języka w typowych zastosowaniach. Jego zadaniem jest ułatwienie życia programiście i pomaganie mu w unikaniu błędów.

Warto również pamiętać, że Java została zaprojektowana w taki sposób, żeby niemożliwe było popełnianie podstawowych błędów związanych z dynamicznym tworzeniem obiektów, które było zmorą początkujących programistów C++. Możliwość wycieku pamięci została zlikwidowana przez zwolnienie programisty z konieczności zarządzania pamięcią. Robi to teraz wirtualna Javy. Natomiast problem wskaźników został rozwiązany w taki sposób, że w Javie ich po prostu nie ma.

W dalszej części rozdziału przygotujemy klasę `Ulamek`, która implementuje ułamek zwykły znany ze szkoły podstawowej. Prześledzimy od początku krok po kroku wszystkie etapy tworzenia klasy od deklarowania pól prywatnych, ich metod dostępowych i konstruktorów poprzez definiowanie przykładowych operacji, jakie można wykonać na ułamku, aż po definiowanie metod służących do konwersji ułamka w łańcuch i liczbę rzeczywistą. Przybliżę przy tym podstawowe pojęcia związane z klasami i projektowaniem obiektowym.

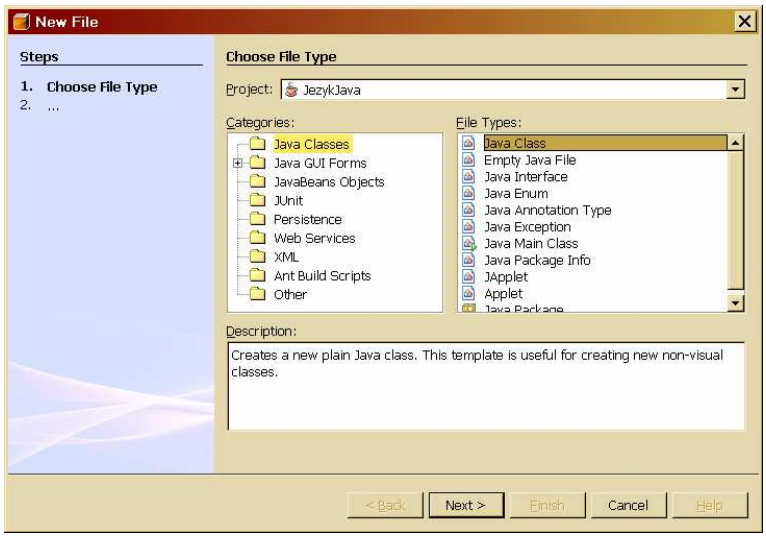
Projektowanie klasy

Do stworzenia klasy możemy użyć kreatora klasy. Kreator zadba o to, żeby nazwa klasy była zgodna z nazwą pliku oraz żeby położenie pliku w strukturze katalogów odpowiadało jego nazwie pakietu. Ponadto stworzy jeszcze za nas szkielet klasy. Nie jest to oczywiście żaden wyczyn, bo deklaracja pustej klasy to tylko pięć słów i kilka nawiasów, ale i tu zdarzyć się może sporo błędów.

Ćwiczenie 2.11.

Aby stworzyć klasę `Ulamek` w pakiecie `JezykJava` umieszczoną w osobnym pliku:

1. Najpierw tworzymy plik, w którym umieścimy klasę. Z menu *File* wybieramy pozycję *New File…*.
2. W oknie *New File* (rysunek 7) wybieramy *Java Classes* w *Categories* i *Java Class* w *File Type*.



Rysunek 7. Kreator klasy

3. Klikamy *Next >*.
4. W drugim kroku kreatora ustalamy nazwę pliku (pole edycyjne *Class Name*): `Ulamek`.

5. Klikamy `Finish`.

<<koniec ćwiczenia>>

Pomijając komentarze w pliku znajdzie się następujący kod:

```
package jezykjava;

public class Ulamek {

    public Ulamek() {
    }

}
```

Plik źródłowy Java powinien rozpoczynać się od deklaracji pakietu, do którego należy klasa. Jeżeli w pliku nie ma deklaracji pakietu — kompilator Java założy, że plik należy do pakietu „nienazwanego”. Wówczas plik źródłowy powinien znajdować się w katalogu bieżącym względem miejsca uruchomienia kompilatora. My jednak zdefiniujemy pakiet, a więc rozpoczynamy plik *Ulamek.java* od polecenia:

```
package jezykjava;
```

Nazwa pakietu musi być w tej sytuacji zgodna z nazwą podkatalogu *jezykjava*.

Po niej powinna znajdować się deklaracja klasy, której nazwa odpowiada (włącznie z wielkością liter) nazwie rdzenia nazwy pliku źródłowego. Poza deklaracją klasy, komentarzem i poleceniami importu bibliotek nie można w tym miejscu umieścić nic innego, bo w Javie nie można deklarować funkcji lub zmiennych poza klasami. Nie ma funkcji i zmiennych globalnych, jakie znamy z C++. Są tylko metody i pola, elementy klas. Zadeklarujmy zatem pustą klasę `Ulamek`:

```
public class Ulamek
{
}
```

Zadeklarowaliśmy klasę publiczną (słowo kluczowe), tj. widoczną także poza plikiem, w którym znajduje się jej definicja. W pliku może być wiele klas, ale tylko jedna z nich może być publiczna i do tego jej nazwa musi odpowiadać rdzeniowi nazwy pliku, także uwzględniając wielkość liter. Jeżeli chcemy nazwać klasę `Ulamek`, jej plik powinien nazywać się *Ulamek.java*.

Aby rozwijać i testować tę klasę, potrzebujemy środowiska do jej testowania. Przygotujmy wobec tego aplet, który będzie do tego służył. Do tego celu posłużymy się już narzędziami dostępnymi w NetBeans.

Środowisko testowania klasy

Za chwilę stworzymy aplikację pełniący rolę środowiska testowania klasy `Ulamek`. Na jej oknie umieścimy listę `javax.swing.JList`, w której można umieszczać informacje o testowaniu klasy, oraz przycisk `javax.swing.JButton` służący jako spust metody testującej, której rolę będzie pełnić jego domyślna metoda zdarzeniowa.

Ćwiczenie 2.12.

Aby stworzyć aplet służący jako środowisko testowania klasy `Ulamek`:

1. Tworzymy nowe okno *Java GUI Forms*, *JFrame Form* o nazwie `UlamekDemo`.
2. Przygotujmy prosty interfejs:
   a) przechodzimy na zakładkę *Desing* edytora,
   b) na aplecie umieszczamy dwa komponenty z palety *Swing*: listę `JList` oraz przycisk `JButton`,
   c) usuwamy całą zawartość listy (własność `model`).
   d) tworzymy metodę zdarzeniową związaną z kliknięciem przycisku (zdarzenie `actionPerformed`).
3. Importujemy klasy pakietu Swing, a więc dodajemy do pliku instrukcję:

```
import javax.swing.*;
```

4. Definiujemy prywatne pole klasy okreslające zawartość listy:

```
private DefaultListModel listModel=new DefaultListModel();
```

5. W konstruktorze model listy przypisujemy do komponentu `jList1`:

```
public UlamekDemo() {
        initComponents();
        jList1.setModel(listModel);
    }
```

6. Dla wygody uruchamiania aplikacji testującej do metody `main` klasy głównej `Main` dodajmy polecenie:

```
new UlamekDemo().setVisible(true);
```

<<koniec ćwiczenia>>

Ta metoda będzie naszym miejscem do testowania klasy `Ulamek`. Ewentualne komunikaty będziemy umieszczać w liście umieszczonej w oknie).

Referencje. Kilka uwag dla programistów C++

Ćwiczenie 2.13.

Aby zadeklarować referencję `ulamek` do obiektu `Ulamek`:

Wewnątrz metody zdarzeniowej `jButton1ActionPerformed` apletu `UlamekDemo`, którą będziemy nazywać po prostu metodą testową, deklarujemy zmienną typu `Ulamek`:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{
    Ulamek ulamek;
}
```

<<koniec ćwiczenia>>

Oto kilka ważnych uwag dotyczących tej referencji (szczególnie dla osób znających C++). Najważniejsza z nich jest taka, że tak zadeklarowana zmienna `ulamek` to w Javie nie jest obiekt. W C++ taka deklaracja oznaczałaby statyczne utworzenie obiektu w obszarze pamięci o nazwie stos. Co więcej, tak utworzony obiekt po wyjściu z metody zdarzeniowej zostałby automatycznie usunięty z pamięci. W Javie to nie jest obiekt. Więcej, w Javie nie ma możliwości statycznego definiowania obiektów. Można to robić jedynie dynamicznie korzystając z operatora `new`. To, co zadeklarowaliśmy powyżej, w Javie jest tylko i wyłącznie referencją do klasy `Ulamek`.

Referencja, znana już w C++, w Javie zyskała na znaczeniu. Referencję można rozumieć jako coś w rodzaju „odsyłacza” do obiektu. Czymś, co może wskazywać na obiekt. Nie jest jednak wskaźnikiem w takim sensie, jakie ma to słowo w C++. W C++ referencja i wskaźnik, choć podobne, są mimo wszystko czymś różnym. Wskaźnik po prostu przechowuje adres pamięci, w której znajduje się obiekt lub wartość zmiennej. Referencja natomiast jest w C++ raczej synonimem lub „etykietą zastępczą”. W Javie nie ma wskaźników, są tylko referencje podobne do tych, jakie znamy z C++.

Mówiąc krótko, zadeklarowaliśmy referencję do klasy typu `Ulamek`, która może wskazywać na ewentualny obiekt, ale na razie nie wskazuje na nic. Nie jest nawet zainicjowana. A powinna.

Ćwiczenie 2.14.

Aby jawnie określić wartość `null` referencji ulamek:

Poprawiamy deklarację referencji `ulamek` w następujący sposób:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{
    Ulamek ulamek = null;
}
```

<<koniec ćwiczenia>>

Referencja `null` nadal nie wskazuje na żaden obiekt, ale jest zainicjowana. Dzięki temu można np. sprawdzić jej wartość za pomocą instrukcji `if (ulamek==null)..`. Bez tego uzyskalibyśmy błąd już w momencie kompilacji informujący, że referencja nie jest zainicjowana.

Tworzenie obiektów

Jeżeli chcemy stworzyć obiekt klasy `Ulamek`, musimy posłużyć się operatorem `new`. Wartością zwracaną przez ten operator jest w Javie referencja do nowo utworzonego obiektu. Możemy zatem zapisać ją do zmiennej `ulamek`.

Ćwiczenie 2.15.

Aby stworzyć obiekt klasy `Ulamek`:

Uzupełniamy metodę zdarzeniową o wyróżniony fragment kodu:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{
    Ulamek ulamek=null;
    ulamek = new Ulamek();
}
```

<<koniec ćwiczenia>>

Podobnie jak w C++, za nazwą klasy następującej po operatorze `new` podajemy argumenty konstruktora (lub jednego z konstruktorów). Jeżeli w klasie nie ma konstruktora — kompilator tworzy, podobnie w C++, bezargumentowy konstruktor domyślny, który wywołaliśmy powyżej.

Ponieważ referencja jednoznacznie wskazuje na obiekt, częstą praktyką jest traktowanie tych dwóch pojęć jako synonimów. Tzn. nazywanie referencji `ulamek` obiektem, mając na myśli obiekt, który ta referencja wskazuje. To pozwala na prostsze omawianie struktury programu i również po tym rozdziale, gdy Czytelnik oswoi się już z Javą, ja też nie będę zbytnio podkreślał różnicy między obiektem a referencją do niego.

Zazwyczaj deklarację referencji i tworzenie obiektu zapisuje się w jednej linii.

Ćwiczenie 2.16.

Aby umieścić deklarację referencji i polecenie stworzenia w jednej linii:

Modyfikujemy metodę zdarzeniową w następujący sposób:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
  {
    Ulamek ulamek = new Ulamek();
  }
```

<<koniec ćwiczenia>>

Sprawdźmy, co może nasz obiekt. Klasa `Ulamek` tylko pozornie jest pusta i bezużyteczna. Otóż wbrew temu, co widzimy w jej definicji, nie jest to klasa, która nie posiada żadnych metod. Możemy to pokazać w bardzo prosty sposób.

Ćwiczenie 2.17.

Aby za pomocą mechanizmu uzupełniania kodu NetBeans przyjrzeć się zawartości klasy `Ulamek`:

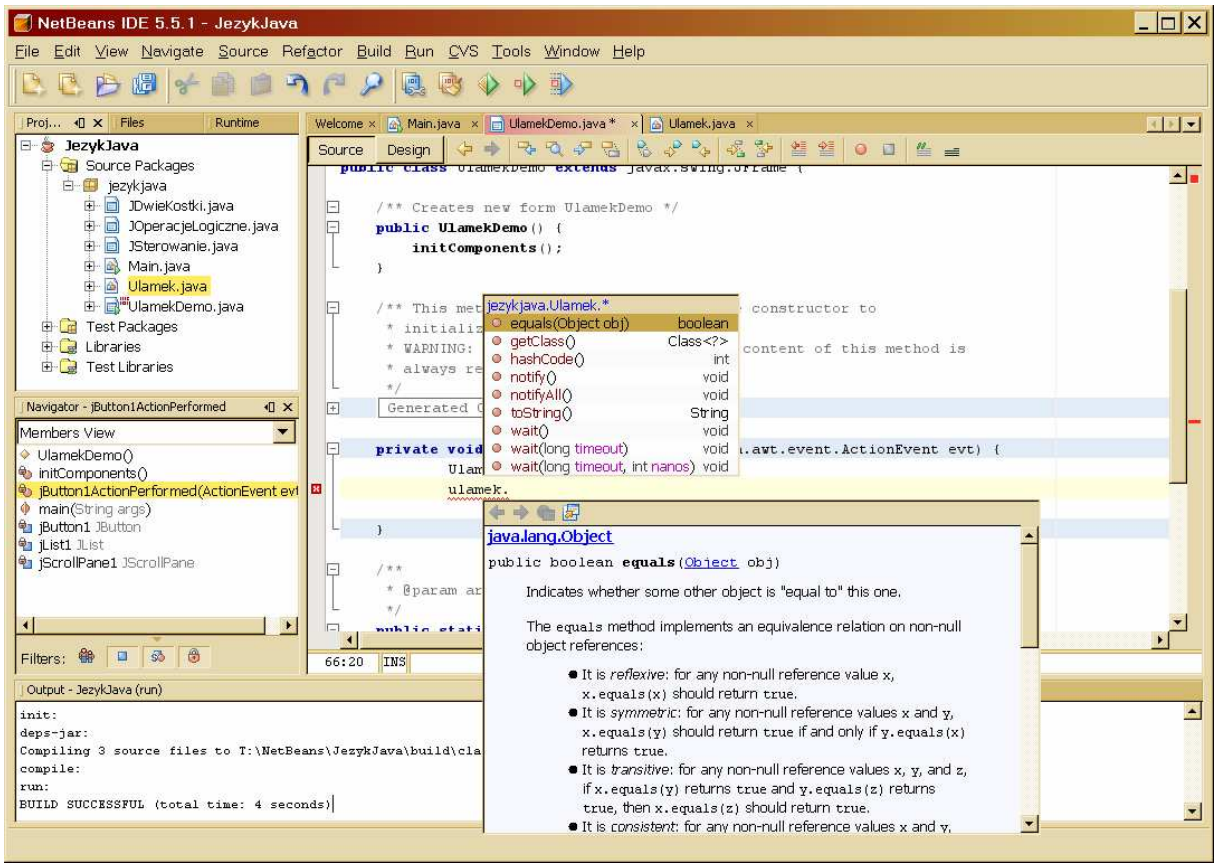
Pod linią definiującą obiekt `ulamek` typu `Ulamek` umieśćmy wywołanie referencji `ulamek` i dodajmy kropkę, tak jakbyśmy chcieli odwołać się do jakiejś metody lub pól[12].

[12] W Javie dostęp do metod oraz pól obiektu i klasy uzyskuje się za pomocą kropki. Nie ma wskaźników, więc znikł też znany z C++ operator `->`.

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
    {
        Ulamek ulamek = new Ulamek();
        ulamek.
    }
```

<<koniec ćwiczenia>>

NetBeans rozwinie listę możliwych do użycia metod pokazaną na rysunku 2.5 (jeżeli tego nie zrobi, trzeba nacisnąć kombinację klawiszy *Ctrl+Space*). Jak widać, jest tego sporo, a więc nasz obiekt pomimo ubogiej definicji jest całkiem bogaty. Dzieje się tak, ponieważ klasa `Ulamek` dziedziczy z klasy `java.lang.Object`. Jeżeli nie wskażemy jawnie klasy bazowej, tj. klasy, z której chcemy, aby nasza klasa dziedziczyła pola i metody, klasą bazową zostaje uznana klasa `java.lang.Object`[13]. W ten sposób każda klasa Javy dziedziczy bezpośrednio lub pośrednio z tej klasy. W konsekwencji każda klasa posiada metody, które w tej chwili oglądamy w liście rozwiniętej przez NetBeans.



Rysunek 2.5. Klasa Ulamek dziedziczy z klasy Object

Ćwiczenie 2.18.

Aby sprawdzić działanie metody `toString` obiektu `ulamek`:

1. Z rozwiniętej w poprzednim ćwiczeniu listy metod i właściwości dostępnych dla obiektu klasy `Ulamek` wybierzmy metodę `toString`.
2. Łańcuch zwrócony przez tę metodę umieśćmy na liście `list1` umieszczając uzyskane w poprzednim punkcie wywołanie metody `ulamek.toString` w argumencie metody `list1.add`:

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
      {
```

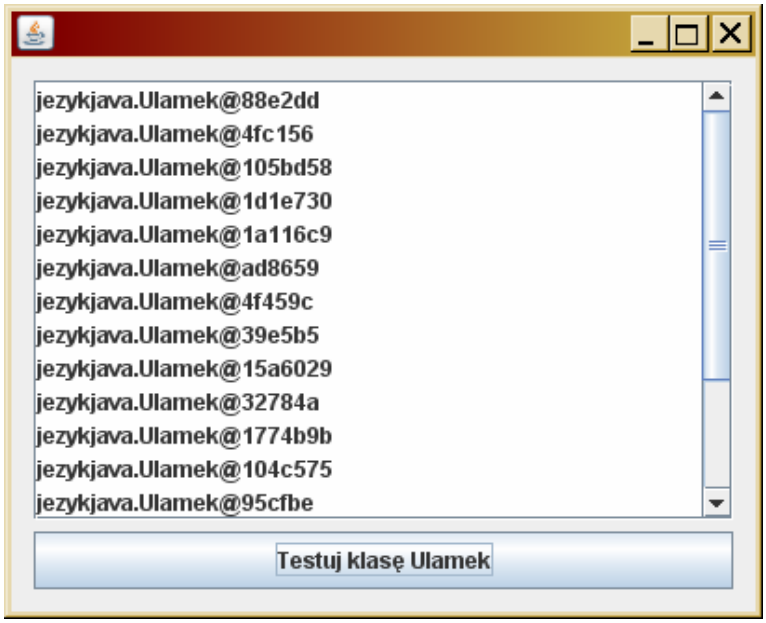
[13] Dokładnie tak samo jak w ObjectPascalu (Delphi). Tam również każdy obiekt bez jawnie wskazanej klasy bazowej dziedziczy z klasy TObject.

```
            Ulamek ulamek = new Ulamek();
            listModel.addElement(ulamek.toString());
        }
```

<<koniec ćwiczenia>>

Z prawej strony listy rozwiniętej po naciśnięciu kropki (rysunek 2.5) pokazany jest typ pola lub typ wartości zwracanej przez metodę. W przypadku metody `toString` jest to łańcuch `String`. Metoda `toString` klasy `Object` tworzy łańcuch składający się z nazwy pakietu, klasy i adresu w pamięci, w którym znajduje się obiekt.

Po skompilowaniu i uruchomieniu apletu `UlamekDemo` (klawisz *F9*), jeżeli naciśniemy przycisk z etykietą *button1*, zobaczymy w liście napis podobny do tego: `jezykjava.Ulamek@b9b538` (zobacz rysunek 9). Po każdym naciśnięciu przycisku adres się zmienia, ponieważ tworzony jest nowy obiekt.



Rysunek 9. Efekt metody toString zdefiniowanej w klasie Object wywołanej na rzecz obiektu klasy Ulamek

Wyciekanie pamięci w C++ i odśmiecacz w Javie

Jednym z najważniejszych powodów usunięcia wskaźników z Javy był bardzo powszechny w programach pisanych w C++ błąd nazywany wyciekiem pamięci. Jeżeli powyższa metoda byłaby metodą napisaną w C++ popełnilibyśmy właśnie taki błąd. Istotą tego błędu jest to, że metoda taka jak nasza napisana w C++:

```
//niepoprawna metoda w C++
void UlamekDemo::metoda()
{
  Ulamek* ulamek=new Ulamek();
}
```

umieszcza stworzony przez operator `new` obiekt na stercie, która nie jest czyszczona po wyjściu z metody. Ale referencja `ulamek` jest zwykłą zmienną, więc znajduje się na stosie i po wyjściu z metody zostaje usunięta. W ten sposób stracimy informacje o położeniu obiektu w pamięci i tym samym tracimy dostęp do niego. Obiekt ten w C++ nie mógłby być usunięty (operator `delete` wymaga podania adresu obiektu) i do momentu zakończenia działania programu będzie tylko zajmował miejsce w pamięci. W ten sposób porcja pamięci wycieknie spod naszej kontroli. Rzeczywisty problem pojawi się jednak dopiero wówczas, gdy ta metoda będzie wywoływana wiele razy, np. w jakiejś pętli. Wtedy wyciek może zmienić się w prawdziwy wylew.

Prawidłowa postać tej metody powinna usuwać obiekt ze sterty za pomocą operatora `delete` przed jej zakończeniem tj.

```
//poprawna metoda w C++
void UlamekDemo::metoda()
```

```
{
  Ulamek* ulamek=new Ulamek();
  delete ulamek;
}
```

Na tym nie kończą się problemy z dynamicznym tworzeniem obiektów w C++. Możliwy jest na przykład taki błąd, w którym zastosujemy operator `delete` usuwający obiekt z pamięci, a po tym próbujemy użyć wskaźnik, który związany był z tym obiektem, tj. wskazujący nadal na to samo miejsce w pamięci. To, co tam jest w chwili wykorzystania wskaźnika, jest trudne do przewidzenia i raczej nie nadaje się do użycia. Jest to tzw. dziki wskaźnik

Jednak w Javie problem ten został rozwiązany w sposób konsekwentny. Na próżno szukać operatora `delete`. Zamiast niego funkcjonuje mechanizm wirtualnej maszyny Javy nazywany odśmiecaczem (ang. *garbage collector*). Jego działanie jest dość proste. Odśmiecacz przeszukuje całą pamięć dostępną dla wirtualnej maszyny Javy i zaznacza obiekty, do których znajduje aktywne odniesienia (referencje). Po przeszukaniu całej pamięci usuwa te obiekty, które nie zostały zaznaczone. Właśnie to stanie się z obiektem ułamka po zakończeniu metody zdarzeniowej i usunięciu referencji `ulamek`. Zatem częsty błąd z C++ jest jak najbardziej poprawnym sposobem programowania w Javie.

Zwykle programiści przesiadający się z C++ przyzwyczajeni do odpowiedzialności za gospodarkę pamięcią w pisanych przez siebie programach przez jakiś czas narzekają, że nie mają żadnej kontroli nad zwalnianiem pamięci. Jednak już po pewnym czasie cieszą się, że ten ciężar został zdjęty z ich pleców i przejęła go wirtualna maszyna Javy.

Pola. Zakres dostępności

Wróćmy do pliku *Ulamek.java* w edytorze[14] i wewnątrz klasy zadeklarujmy dwa prywatne pola klasy typu `int` o nazwach `licznik` i `mianownik`. Pole `mianownik` inicjujemy wartością 1, a pola `licznik` nie inicjujemy jawnie. Zrobi to zatem kompilator przypisując mu wartość równą 0.

Ćwiczenie 2.19.

Aby zdefiniować pola licznik i mianownik klasy `Ulamek`:

1. Zmieniamy zakładkę na górze okna edytora na *Ulamek.java*.
2. Zmieniamy zakładkę na dole edytora na *Source*.
3. Uzupełniamy kod klasy o wyróżnioną w poniższym kodzie linię:

```
public class Ulamek {

    private int licznik, mianownik=1;

    /** Creates a new instance of Ulamek */
    public Ulamek() {
    }

}
```

<<koniec ćwiczenia>>

Pola klasy zostały zdefiniowane z dodatkowym słowem kluczowym `private` (z ang. *prywatne*). Co oznacza, że pole jest prywatne? Dokładnie tyle, że nie jest widoczne na zewnątrz klasy. Nie zobaczymy jej więc w liście dostępnych metod i pól po naciśnięciu kropki w aplecie `UlamekDemo`. Takie pole można czytać i modyfikować tylko i wyłącznie poleceniami umieszczonymi wewnątrz klasy. Dla odmiany pole publiczne jest dostępne z

[14] Kolejne ćwiczenia będą wymagały ciągłego przełączania między klasą `Ulamek` i apletem służącym do jego testowania `UlamekDemo`. Obie klasy powinny być dostępne na górnych zakładkach w oknie edytora. Przed wykonaniem modyfikacji kodu proponowanych w dalszych ćwiczeniach należy za każdym razem zwrócić uwagę na to, której klasy ona dotyczy.

zewnątrz i możemy się do niego odwoływać korzystając ze zwykłego zapisu, tj. `ulamek.pole`. Jest jeszcze zakres chroniony pól i metod klasy (identyfikowany słowem kluczowym `protected`), ale o nim opowiem niżej przy okazji dziedziczenia.

Metody. Referencja this

Wewnątrz klasy możemy posługiwać się referencją `this`, żeby odwoływać się do pól i metod bieżącej klasy. Zazwyczaj nie jest to potrzebne, bo nazwy zmiennych automatycznie interpretowane są jako pola bieżącej klasy, ale czasem zdarzają się takie sytuacje, jedną z nich zaraz przygotuję niżej, że użycie referencji `this` jest konieczne.

Jak dotąd klasa `Ulamek` ma tylko dwa prywatne pola: `licznik` i `mianownik`. Ponieważ są one prywatne, ewentualny użytkownik klasy nie ma możliwości żadnego wpływu na ich wartość. Aby móc ustalić lub odczytać ich wartość „z zewnątrz klasy", musimy dodać metody publiczne, które pozwolą na zmianę i odczytanie ich wartości. Będziemy nazywać je metodami dostępowymi, ponieważ pozwalają na dostęp z zewnątrz do prywatnych pól klasy.

Dodajmy najpierw do klasy `Ulamek` dwie metody `setLicznik` i `getLicznik`, które pozwolą na zmianę i odczytanie wartości pierwszego z jej pól — pola `licznik`. Zadaniem metody `setLicznik` będzie ustalenie nowej wartości pola `licznik` zgodnie z wartością podaną w argumencie metody. Metoda ta powinna zatem przyjmować argument typu `int` o nazwie `licznik` (zbieżność nazw pola klasy i argumentu metody wprowadziłem specjalnie, aby konieczne było wykorzystanie referencji `this`, w ogólności nie jest ona konieczna). Druga metoda o nazwie `getLicznik`, pozwalająca na sprawdzenie wartości pola `licznik`, powinna zwracać wartość typu `int`.

Ćwiczenie 2.20.

Aby dodać do klasy `Ulamek` dwie metody `setLicznik` i `getLicznik` związane z polem licznik:

Dodajemy do definicji klasy następujące metody:

```
public class Ulamek
{
  private int licznik, mianownik=1;

  public void setLicznik(int licznik) {this.licznik=licznik;};
  public int getLicznik() {return licznik;}
}
```

<<koniec ćwiczenia>>

Przyjrzyjmy się pierwszej z nich:

```
public void setLicznik(int licznik)
{
  this.licznik=licznik;
};
```

Jest to metoda publiczna niezwracająca wartości, co oznaczamy wstawiając typ `void` (z ang. *pusty*, *nieważny*). Jej argumentem jest liczba typu `int` lokalnie w tej metodzie dostępna w zmiennej o nazwie `licznik` zadeklarowanej w sygnaturze metody. Ale przecież taką samą nazwę ma prywatne pole klasy. Tak właśnie wygląda typowa sytuacja, w której trzeba użyć referencji `this`. Ponieważ lokalnie zadeklarowana zmienna `licznik` przesłoniła pole klasy, do pola możemy dostać się tylko korzystając z odwołania postaci `this.licznik`. Wówczas kompilator wie, że chodzi o pole bieżącej klasy, a nie o lokalną zmienną.

Teraz spójrzmy na drugą metodę:

```
public int getLicznik()
{
  return licznik;
}
```

Ta metoda również jest publiczna, lecz tym razem zwraca wartość typu `int`. Jeżeli metoda zwraca jakąś wartość, to wewnątrz niej musi znaleźć przynajmniej jedno słowo kluczowe `return`, które zostanie na pewno wywołane („na pewno", czyli nie umieszczone za instrukcją `if` itp.). Za takim pewnym wywołaniem `return` nie mogą już znajdować się żadne polecenia, ponieważ zgłoszony zostanie przez kompilator błąd sygnalizujący, że w metodzie znajduje się niedostępny kod.

W następnym kroku do klasy dodamy metody pozwalające na modyfikację i odczytywanie wartości pola `mianownik`. Podczas ich definiowania musimy pamiętać, żeby nie pozwolić na sytuację, w której wartość tego pola zmieniana jest na zero.

Ćwiczenie 2.21.

Aby z polem mianownik związać metody `setMianownik` i `getMianownik`:

Do klasy dodajemy definicje dwóch metod:

```
public class Ulamek
{
  private int licznik, mianownik=1;

  public void setLicznik(int licznik) {this.licznik=licznik;};
  public int getLicznik() {return licznik;}
  public void setMianownik(int mianownik)
   {
      if (mianownik!=0) this.mianownik=mianownik;
    };
  public int getMianownik() {return mianownik;}
}
```

<<koniec ćwiczenia>>

Metoda ustalająca wartość pola `mianownik`

```
public void setMianownik(int mianownik)
{
  if (mianownik!=0) this.mianownik=mianownik;
};
```

sprawdza, czy podana w argumencie wartość jest różna od zera i przypisuje nową wartość pola `mianownik` tylko i wyłącznie w takim przypadku.

*

Zamiast pozwolić użytkownikowi klasy modyfikować bezpośrednio pola `licznik` i `mianownik`, udostępniliśmy mu jedynie metody, które na to pozwalają. Jest z tym znacznie więcej roboty (zamiast jednej linii kodu deklarującej publiczne właściwości mamy wiele dodatkowych linii definicji metod dostępowych), ale od razu widać zaletę tego typu podejścia. Przy ustalaniu właściwości `mianownik` mamy możliwość kontrolowania przypisywanej wartości i zareagowania, jeżeli jest ona równa 0. To daje możliwość uniknięcia błędów, jakie mogłyby się pojawić w trakcie działania programu, spowodowanych nieprzewidywalnymi wartościami przypisanymi polom przez otoczenie klasy `Ulamek`.

Sprawdźmy działanie zdefiniowanych w dwóch poprzednich ćwiczeniach metod klasy `Ulamek` w naszym środowisku testowania, tj. w aplecie `UlamekDemo`.

Ćwiczenie 2.22.

Aby sprawdzić w aplecie `UlamekDemo` działanie metod `set..` i `get..`:

1. Zmieniamy zakładkę edytora (w górnej części okna edytora) na *UlamekDemo*.
2. Modyfikujemy metodę zdarzeniową umieszczając w niej wyróżniony kod:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        Ulamek ulamek = new Ulamek();
        ulamek.setLicznik(1);
        ulamek.setMianownik(2);
        listModel.addElement(""+ulamek.getLicznik()+"/"+ulamek.getMianownik());
    }
```

3. Kompilujemy i uruchamiamy aplikację `UlamekDemo` (klawisz *F6*).
4. Po uruchomieniu aplikacji klikamy przycisk.

<<koniec ćwiczenia>>

Po stworzeniu obiektu w pierwszej linii metody zdarzeniowej wykorzystujemy jego metody `set..` do ustalenia wartości pól `licznik` i `mianownik`, a następnie dodajemy do listy `list1` łańcuch postaci *licznik / mianownik*, który w tym przypadku będzie równy 1/2. Skorzystaliśmy przy tym z omówionej wyżej sztuczki, która pozwala uniknąć jawnej konwersji liczb typu `int` zwracanych przez metody `get..` w łańcuchy.

Ćwiczenie 2.23.

Aby sprawdzić, co się stanie, jeżeli wykonamy metodę `setMianownik` z argumentem równym zero:

1. Metodę zdarzeniową w aplecie `UlamekDemo` należy zmodyfikować w następujący sposób:

```
void button1_actionPerformed(ActionEvent e)
  {
    Ulamek ulamek = new Ulamek();
    ulamek.setLicznik(1);
    ulamek.setMianownik(0);
    list1.add(""+ulamek.getLicznik()+"/"+ulamek.getMianownik());
  }
```

2. Kompilujemy i uruchamiamy aplet.
3. Po uruchomieniu naciskamy przycisk.

<<koniec ćwiczenia>>

Próba ustalenia wartości mianownika równej zero nie powiodła się, jego wartość pozostanie równa wartości zainicjowanej przy deklarowaniu właściwości i w liście zobaczymy łańcuch 1/1.

Zgłaszanie błędów w metodach za pomocą wyjątków

Metoda `setMianownik` klasy `Ulamek` sprawdza, czy podana w argumencie wartość jest właściwa, ale w przypadku błędnej nie zgłasza tego faktu użytkownikowi. A powinna, bo programista używający naszej klasy może nie być świadomy popełnianego błędu. Można to zrobić na kilka sposobów. WinAPI to zmienna globalna, do której zapisywana jest informacja w razie wystąpienia błędu w metodzie. Można również dodać do metody zmienną publiczną `error`, którą użytkownik klasy może sprawdzać po wywołaniu metod. Są to jednak metody, które wymagają od programisty aktywności, podczas gdy wiadomo, że programiści to grupa raczej leniwa (szczególnie, że prowadzą nocny i siedzący tryb życia).

Dlatego powstała nowa technika obsługi błędów, w której aktywność jest całkowicie po stronie klasy i polega na zgłaszaniu wyjątków, tj. informacji o sytuacjach wyjątkowych. Zastosujmy zatem tę technikę w metodzie `setMianownik`. Najpierw jeszcze tylko trochę ideologii.

Czym dokładnie jest wyjątek (ang. *exception*)? Wyjątek to obiekt, który jest zgłaszany[15] przez metodę w momencie popełnienia błędu, a który pełni po prostu rolę nośnika informacji o błędzie. Obiekt wyjątku w Javie zgłaszany przez metodę musi być typu `Exception` lub z niego dziedziczącym. Można tworzyć własne klasy wyjątków, co ma sens, jeżeli poza komunikatem chcemy przekazać jakieś szczególne informacje o stanie

[15] W angielskim funkcjonują w tym kontekście terminy *throw*, który oznacza dokładnie rzucać, i *raise*, czyli wznosić (np. protest lub obiekcję).

metody w momencie wystąpienia błędu. Tu jednak skorzystamy tylko z klasy `Exception` ustalając za pomocą odpowiedniego konstruktora treść komunikatu o błędzie.

Ćwiczenie 2.24.

Aby zmodyfikować metodę `setMianownik` w taki sposób, że po uruchomieniu jej z argumentem o wartości 0 zgłoszony zostanie wyjątek z komunikatem `Mianownik nie może być równy 0`:

1. Wracamy do edycji klasy `Ulamek` (plik *Ulamek.java*), tj. zmieniamy zakładkę na górze okna edytora na *Ulamek*.
2. Wprowadzamy następujące modyfikacje do metody `setMianownik`:

```
public void setMianownik(int mianownik) throws Exception
    {
      if (mianownik!=0) this.mianownik=mianownik;
      else throw new Exception("Mianownik nie może być równy 0");
    };
```

<<koniec ćwiczenia>>

Zgłaszanie wyjątku odbywa się za pomocą słowa kluczowego `throw`. Po nim następuje referencja do obiektu. Tutaj tworzymy obiekt za pomocą operatora `new` w miejscu jego wykorzystania do zgłoszenia wyjątku. Każda metoda, która zgłasza wyjątek, musi posiadać w sygnaturze klauzulę deklarującą ją jako metodę, która może zgłosić dany typ wyjątku. Stąd dodatkowy człon w sygnaturze ze słowem kluczowym `throws`.

Obsługa wyjątków

W sprawie wyjątków Java jest konsekwentna i zasadnicza — każde wywołanie metody, która zadeklarowana jest jako taka, która może zgłosić wyjątek, musi być otoczone konstrukcją `try...catch`, która służy do wyłapywania i obsługi wyjątków. W innym przypadku otrzymamy błąd kompilatora sygnalizujący, że nie została obsłużona możliwość zgłoszenia przez metodę wyjątku. Nie ma więc sytuacji, w której programista nie zostanie poinformowany o wystąpieniu błędu.

Ćwiczenia 2.25.

Aby w metodzie zdarzeniowej apletu `UlamekDemo` dodać obsługę wyjątku wokół metody `setMianownik`:

1. Wracamy do edycji klasy apletu `UlamekDemo` — przełączamy zakładkę na górze edytora na *UlamekDemo*.
2. Po zmianach wprowadzonych w metodzie `setMianownik` jej uruchomienie wymaga otoczenia konstrukcją `try...catch`. W naszym przypadku może to wyglądać następująco:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
   Ulamek ulamek = new Ulamek();
   ulamek.setLicznik(1);
   try
   {
      ulamek.setMianownik(0);
      listModel.addElement(""+ulamek.getLicznik()+"/"+ulamek.getMianownik());
   }
   catch(Exception exc)
   {
      listModel.addElement("Błąd: "+exc.getMessage());
   }
}
```

3. Uruchamiamy aplet (*F6*) i naciskamy przycisk uruchamiający metodę testującą klasę `Ulamek`.

<<koniec ćwiczenia>>

Dlaczego wywołanie metody `listModel.addElement` pokazującej wartość ułamka znajduje się wewnątrz `try`, podczas gdy nie wymaga wcale obsługi wyjątku? Czy można ją przenieść na koniec metody zdarzeniowej? Metoda prezentująca wartość ułamka znajduje się wewnątrz sekcji `try` dlatego, że w ten sposób w przypadku zgłoszenia wyjątku przez metodę `ulamek.setMianownik` nie zostanie wykonana. Po zgłoszenia wyjątku wątek nie wraca już do następnej linii w sekcji `try`. Zostaje wykonana zawartość sekcji `catch` i wątek przechodzi do linii następującej po `catch`. Umieszczenie tej linii za konstrukcją obsługi wyjątków spowodowałoby wobec tego wyświetlenie wartości ułamka w `jList1` nawet wówczas, gdy zmiana mianownika nie powiodła się.

Uwaga! W razie wystąpienia wyjątku i opuszczeniu sekcji `try` wątek już do niej nie powróci.

Do przekazania informacji o błędzie wykorzystaliśmy metodę `Exception.getMessage`, która zwraca dokładnie ten sam łańcuch, który był podany na przechowanie obiektowi-nośnikowi w argumencie konstruktora.

Konstruktor

Szczególną metodą każdej klasy jest metoda o nazwie identycznej jak nazwa klasy (w naszym przypadku metoda `Ulamek.Ulamek`). Jest to tzw. konstruktor. Konstruktor jest wywoływany w momencie tworzenia obiektu i jest wykorzystywany do jego inicjacji, a dokładniej do inicjacji jego pól.

Konstruktor inicjujący pola obiektu typu `Ulamek` powinien korzystać z przygotowanych wcześniej metod `setLicznik` i `setMianownik`. Po pierwsze dlatego, że kod nie powinien być powtarzany, bo utrudnia to pilnowanie spójności klasy w przypadku jej modyfikacji. A po drugie dlatego, że wymusza to odpowiednią obsługę wyjątków.

Ćwiczenie 2.26.

Aby dodać do klasy `Ulamek` dwuargumentowy konstruktor pozwalający na ustalenie wartości licznika i mianownika:

1. Wróćmy do edycji klasy `Ulamek`.
2. Konstruktor domyślny (bezargumentowy) pozostawiamy bez zmian.
3. Dodajemy natomiast nową definicję konstruktora klasy `Ulamek`, króry inicjuje stan obiektu zgodnie z podanymi warościami licznika i mianownika:

```
public Ulamek(int licznik,int mianownik) throws Exception
  {
    try
    {
      setMianownik(mianownik);
      setLicznik(licznik);
    }
    catch(Exception exc)
    {
      throw exc;
    }
  }
```

<<koniec ćwiczenia>>

Konstruktor jest dwuargumentowy. Jako jego argumenty podajemy wartości typu `int` inicjujące pola `licznik` i `mianownik`. W przypadku gdy wartość mianownika jest równa 0, wywołana z konstruktora metoda `setMianownik` zgłosi wyjątek, którego obsługa w konstruktorze polega jedynie na tym, że zostanie przekazany dalej. W ten sposób konstruktor również musi być zadeklarowany jako metoda zdolna zgłosić wyjątek.

Takie rozwiązanie ma podstawową zaletę — w razie podania nieodpowiedniej wartości mianownika konstruktor nie zostanie wykonany i obiekt nie powstanie. Gdybyśmy w sekcji `catch` konstruktora usunęli dyrektywę

zgłaszania wyjątku, wówczas obiekt by powstał, ale zostałby zainicjowany domyślnymi wartościami 0/1. Jednak nie byłoby to dobre rozwiązanie, ponieważ istnieje groźba, że programista nie zauważy popełnianego błędu.

W sytuacji, w której mamy konstruktor dwuargumentowy, metoda testująca naszą klasę w aplecie `UlamekDemo` musi zostać zmodyfikowana. Zbędne staje się wywołanie metod `setLicznik` i `setMianowik`:

Ćwiczenie 2.27.

Aby wykorzystać konstruktor klasy `Ulamek` w metodzie testującej:

1. Przechodzimy do edycji klasy apletu `UlamekDemo`.
2. Modyfikujemy metodę zdarzeniową zgodnie z wyróżnieniem w następującym kodzie:

```
  private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
          Ulamek ulamek=null;
          try
          {
              ulamek = new Ulamek(1,2);
              listModel.addElement(""+ulamek.getLicznik()+"/"+ulamek.getMianownik());
          }
          catch(Exception exc)
          {
              listModel.addElement("Błąd: "+exc.getMessage());
          }
      }
```

<<koniec ćwiczenia>>

Uwaga! Zgłoszenie wyjątku przez konstruktor blokuje powstanie obiektu.

Jeżeli zadeklarowaliśmy konstruktor (z argumentami lub bez), kompilator nie tworzy konstruktora domyślnego. I dobrze, bo w zasadzie nie powinno być konstruktorów, w których użytkownik klasy nie jest zmuszany do świadomego inicjowania obiektu.

Przeciążanie metod

Konstruktor klasy `Ulamek` jest dobrym przykładem, na którym można zilustrować przeciążanie metod. Przeciążanie metod oznacza tylko tyle, że w klasie deklarujemy wiele metod o tej samej nazwie, ale z różnymi argumentami[16].

Ćwiczenie 2.28.

Aby zdefiniować w klasie `Ulamek` jednoargumentowy konstruktor pozwalający na ustalenie jedynie wartości licznika:

1. Wracamy do edycji klasy `Ulamek`.
2. Dopisujemy do niej poniższą definicję konstruktora:

```
  public Ulamek(int liczba){
     setLicznik(liczba);
  }
```

<<koniec ćwiczenia>>

[16] W Javie nie ma bardzo wygodnych znanych z C++ wartości domyślnych podawanych w deklaracji metod, w tym konstruktorów. Kolejna rzecz to fakt, że zadeklarowanie jednoargumentowego konstruktora z argumentem typu `int` nie oznacza, jak było w C++, umożliwienia jawnej lub niejawnej konwersji z `int` do `Ulamek`.

Konstruktor jednoargumentowy jest znacznie uproszczoną wersją konstruktora z poprzedniego ćwiczenia. W tym przypadku nie ma konieczności deklarowania możliwości zgłaszania wyjątku przez konstruktor, bo inicjacja pola `mianownik` odbywa się niejawnie przez pozostawienie jego domyślnej wartości. Nie istnieje możliwość wywołania konstruktora w innej sytuacji niż przy tworzeniu obiektu, więc nie ma groźby, że mianownik ma w momencie wykonywania tego konstruktora wartość inną niż 1.

Mamy zatem metodę, w tym przypadku konstruktor, która jest dwukrotnie przeciążona. Można jeszcze raz ją przeciążyć tworząc konstruktor domyślny (tj. bezargumentowy), ale rozmyślnie go nie tworzymy — niech programista korzystający z klasy `Ulamek` będzie zmuszony do rozmyślnego zainicjowania obiektów tej klasy.

Przetestujmy nowy konstruktor tworząc w metodzie zdarzeniowej apletu `UlamekDemo` obiekt o nazwie „trzy” korzystając z jednoargumentowego konstruktora.

Ćwiczenie 2.29.

Aby w metodzie testującej wykorzystać konstruktor jednoargumentowy i sprawdzić wartości pól `licznik` i `mianownik`:

1. Ponownie przechodzimy do edycji apletu `UlamekDemo`.
2. Modyfikujemy testową metodę zdarzeniową dopisując wyróżniony w poniższym kodzie fragment:

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
            Ulamek trzy=new Ulamek(3);
            listModel.addElement(""+trzy.getLicznik()+"/"+trzy.getMianownik());

            try
            {
                Ulamek pol = new Ulamek(1,2);
                listModel.addElement("" + pol.getLicznik() + "/" + pol.getMianownik());
            }
            catch(Exception exc)
            {
                listModel.addElement("Błąd: "+exc.getMessage());
            }
        }
```

<<koniec ćwiczenia>>

Wywołanie konstruktora jednoargumentowego, w którym wartość mianownika jest ustalana na 1, jest równoznaczne z tworzeniem obiektu typu `Ulamek`, który przechowuje liczbę całkowitą. Można zatem ten konstruktor traktować jak sposób na konwersję liczb całkowitych do ułamków zwykłych. W C++ oznaczało to, że można stosować od tego momentu operator przypisania do inicjowania obiektów `Ulamek` liczbą całkowitą typu `int`, np. `Ulamek ulamek=3;`. W Javie tak nie jest.

Porównywanie obiektów. Rzutowanie referencji. Nadpisywanie metod

Wykonajmy prosty test porównujący obiekty ułamków. Jeżeli zadeklarujemy w metodzie testowej apletu `UlamekDemo` trzy obiekty typu `Ulamek`:

```
Ulamek u1 = new Ulamek(1,3);
Ulamek u2 = new Ulamek(1,3);
Ulamek u3 = new Ulamek(2,6);
```

i sprawdzimy wartość następujących wyrażeń `u1 == u2` oraz `u1 == u3`, okaże się, że oba są fałszywe. Dlaczego? Ponieważ porównujemy referencje tych obiektów, a nie je same. A ponieważ są to różne obiekty (nie w sensie wartości, ale jako egzemplarze — zajmują różne miejsca w pamięci), ich referencje muszą być różne. Operator porównania jest więc przydatny jedynie wtedy, gdy chcemy sprawdzić, czy dwie referencje wskazują na ten sam obiekt, ale nie do porównania stanu obiektów.

Do porównywania obiektu służy zdefiniowana w klasie `Object` metoda `equals`. Skoro jest zdefiniowana w klasie `Object`, musi być także dostępna w każdej klasie, w tym w `Ulamek` (porównaj ćwiczenie 2.17). Korzystając z niej, sprawdzimy w następnym ćwiczeniu wartości wyrażeń `u1.equals(u1)`, `u1.equals(u2)` i `u1.equals(u3)`.

Ćwiczenie 2.30.

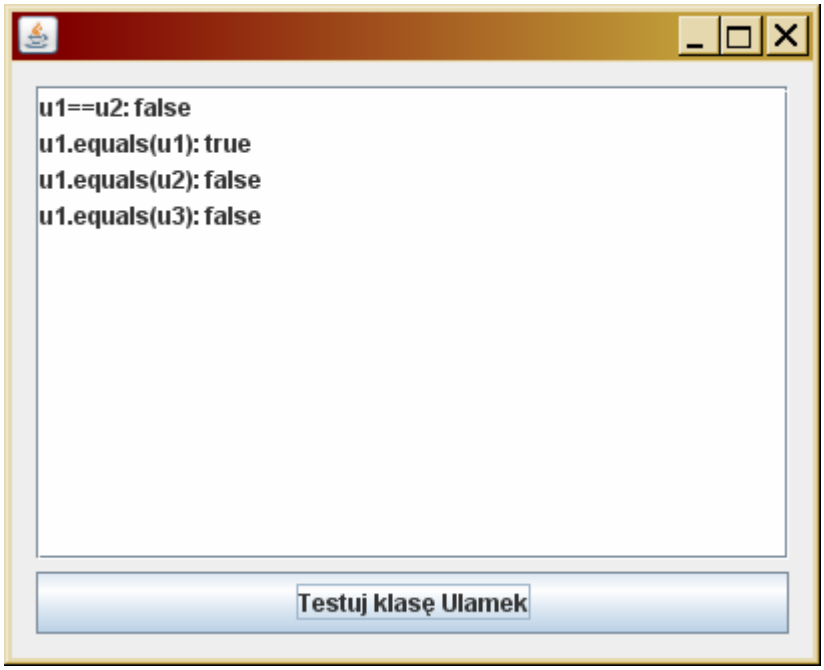
Aby porównać obiekty `u1`, `u2` i `u3` korzystając z metody `equals` klasy `Ulamek`:

1. Nadal edytujemy klasę apletu `UlamekDemo`.
2. W następujący sposób modyfikujemy metodę testującą:

```
  private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
          try
          {
              Ulamek u1 = new Ulamek(1,3);
              Ulamek u2 = new Ulamek(1,3);
              Ulamek u3 = new Ulamek(2,6);
              listModel.addElement("u1==u2: " + (u1==u2));
              listModel.addElement("u1.equals(u1): " + (u1.equals(u1)));
              listModel.addElement("u1.equals(u2): " + (u1.equals(u2)));
              listModel.addElement("u1.equals(u3): " + (u1.equals(u3)));
          }
          catch (Exception exc)
          {
              listModel.addElement("Błąd: "+exc.getMessage());
          }
      }
```

<<koniec ćwiczenia>>

Wynik, który uzyskamy po uruchomieniu apletu i naciśnięciu przycisku, został pokazany na rysunku 10.



Rysunek 10. Testowanie metody equals klasy Ulamek

Tylko porównanie obiektu samego ze sobą za pomocą funkcji `equals` dało wynik pozytywny. Tak bowiem jest zdefiniowana metoda `equals` w klasie `Object` — nie może zatem wiedzieć, w jaki sposób porównać obiekty klasy `Ulamek`. Jeżeli chcemy, aby `equals` dawała lepsze wyniki dla klasy `Ulamek`, musi ją na nowo zdefiniować w tej klasie, czyli „nadpisać” metodę klasy bazowej.

Zanim zaczniemy nadpisywać metodę `equals`, musimy najpierw zdecydować, o jaką równość nam chodzi. Czy o dosłowną równość obiektów, równość ich stanów? W takim przypadku `equals` powinna porównywać pola `licznik` i `mianownik` — i jeżeli są takie same, to wtedy, i tylko wtedy, uznajemy, że obiekty są identyczne, a `equals` zwraca `true`. A może należy brać pod uwagę interpretację tej klasy jako ułamka i sprawdzać, czy wartości ułamków utworzonych z liczników i mianowników porównywanych klas są sobie równe. W pierwszym przypadku porównanie `u1` i `u2` da wynik pozytywny, a `u1` i `u3` — negatywny. W drugim przypadku oba wywołania metody `equals` zwrócą wartość prawdziwą. Proponuję zaimplementować drugą opcję.

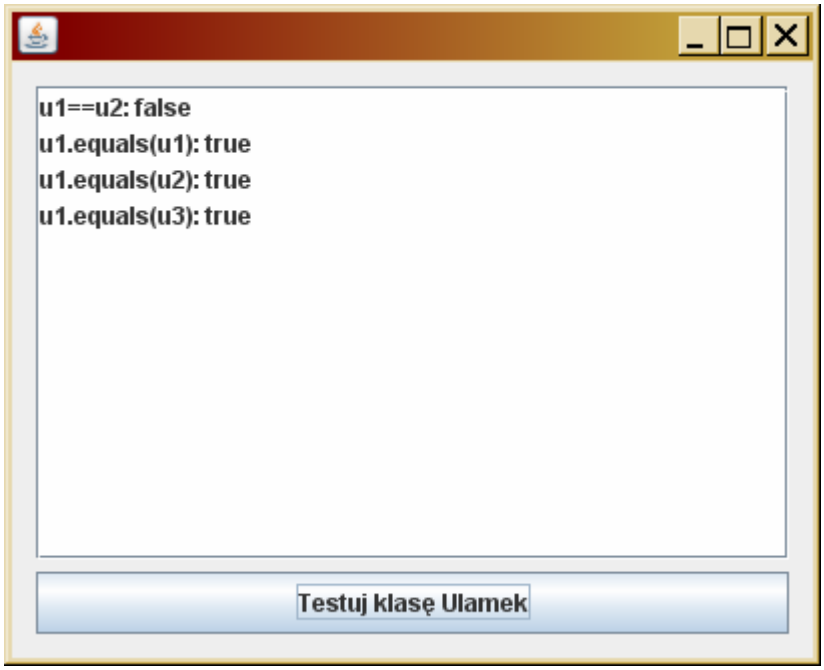
Ćwiczenie 2.31.

Aby nadpisać metodę `equals` w klasie `Ulamek`:

1. Wracamy do edycji klasy `Ulamek`.
2. Deklarujemy w niej metodę `equals` o identycznej sygnaturze jak metoda zdefiniowana w klasie `Object`, tj. `public boolean equals(Object obj)`.
3. Umieszczamy w niej polecenie przyrównania stosunków liczników porównywanych ułamków do stosunku ich mianowników (metoda powinna zwracać wartość logiczną tego przyrównania):

```
public boolean equals(Object obj)
  {
    Ulamek ulamek=(Ulamek)obj;
    return (this.getLicznik()/(double)ulamek.getLicznik() ==
            this.getMianownik()/(double)ulamek.getMianownik());
  }
```

<<koniec ćwiczenia>>



Rysunek 11. Wyniki porównań po zmianie metody Equals

W pierwszej linii metody wykorzystano jawne rzutowanie referencji do klasy `Object` na referencję do klasy `Ulamek`. To jest zawsze możliwe, jeżeli dwie klasy związane są relacją dziedziczenia (i to nie tylko

bezpośredniego) zarówno w dół, jak i w górę (tzn. klasy rozszerzonej na bazową i odwrotnie)[17]. O ile dziedziczenie w dół np.

```
Object obj=(Object)new Ulamek();
```

jest zawsze bezpieczne, bo klasa bazowa musi zawierać podzbiór właściwości i metod klasy rozszerzonej, to rzutowanie wstępujące może być oczywiście niebezpieczne. Dopuszczalna jest następująca konstrukcja:

```
(new Ulamek()).equals(new Object());
```

Wywoływana jest tu metoda `equals` obiektu klasy `Ulamek`. Jej argumentem jest referencja do obiektu klasy `Object`. Wewnątrz metody argument zostanie zrzutowany na obiekt klasy `Ulamek` i podjęta zostanie próba wywołania jego metod `getLicznik` i `getMianownik`, których on oczywiście nie ma. Co się wówczas stanie? Zostanie zgłoszony wyjątek `ClassCastException` (ang. *cast* oznacza rzutowanie). Tak naprawdę jedyny przypadek, gdy może powieść się konwersja argumentu `obj` będącg referencją do klasy `Object` na referencję do klasy `Ulamek`, to ten, gdy `obj` w rzeczywistości odnosi się do obiektu typu `Ulamek` lub z niego dziedziczącego, tzn. gdy mamy do czynienia z sytuacją, którą schematycznie można przedstawić w następującym poleceniu:

```
(new Ulamek()).equals((Object)new Ulamek());
```

Wówczas rzutowanie argumentu na referencję do klasy `Ulamek` wewnątrz metody `equals` na pewno się uda.

Czy obsługiwać ewentualny wyjątek zgłaszany przy rzutowaniu w metodzie `equals` (tzn. czy otoczyć pierwszą linię metody `equals` przez konstrukcję `try..catch`) i nie dopuścić do jego ujawnienia? Ogólnie rzecz biorąc, nie powinno się tego robić. Nie powinno się ukrywać występowania błędów, bo zazwyczaj pojawiają się one przez niedopatrzenie programisty i powinien mieć on możliwość diagnozowania ich wystąpienia.

Można zrobić za to inną rzecz (wszystko zależy od zadań, jakie stawiamy klasie `Ulamek`) — możemy sprawdzić, czy w referencji do klasy `Object` podawanej przez argument metody `equals` kryje się w rzeczywistości obiekt typu `Ulamek`. Jeżeli nie, to po prostu możemy zwracać wartość metody `equals` równą `false`. Oczywiście informacja o takim działaniu metody `equals` powinna znaleźć się w dokumentacji klasy.

Ćwiczenie 2.32.

Aby w metodzie `equals` sprawdzić typ porównywanego obiektu:

Wystarczy dodać na początku definicji metody `equals` jedną linię:

```
public boolean equals(Object obj)
  {
    if (!(obj instanceof Ulamek)) return false;
    Ulamek ulamek=(Ulamek)obj;
    return (this.getLicznik()/(double)ulamek.getLicznik() ==
            this.getMianownik()/(double)ulamek.getMianownik());
  }
```

<<koniec ćwiczenia>>

W przypadku gdy obiekt wskazywany przez referencję `obj` nie jest klasy `Ulamek`, metoda zwraca wartość fałszywą.

Kopiowanie obiektów[18]. Referencja super. Zakres chroniony właściwości

Zajmiemy się teraz kopiowaniem obiektów. Okazuje się, że również tu czyhają na nas pułapki.

Ćwiczenie 2.33.

Aby w metodzie zdarzeniowej apletu `UlamekDemo` stworzyć obiekt `u1` typu `Ulamek`, a następnie zadeklarować referencje `u2` i zainicjować ją wartością odczytaną z `u1`:

---

[17] Rzutowanie w dół na klasę bazową nazywane jest zawężaniem typu.

[18] Osoby znające C++ powinny w tym momencie przypomnieć sobie termin *konstruktor copy*.

1. Przechodzimy do edycji apletu testującego `UlamekDemo`.
2. W metodzie zdarzeniowej znajdującej się w klasie `UlamekDemo` dopisujemy wyróżniony poniżej kod:

```
void button1_actionPerformed(ActionEvent e)
  {
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        try
        {
            Ulamek u1 = new Ulamek(1,3);
            Ulamek u2 = u1;
            u2.setLicznik(3);
            u2.setMianownik(4);
            listModel.addElement("u1: "+u1.getLicznik()+"/"+u1.getMianownik());
            listModel.addElement("u2: "+u2.getLicznik()+"/"+u2.getMianownik());
        }
        catch (Exception exc)
        {
            listModel.addElement("Błąd: "+exc.getMessage());
        }
    }
```

<<koniec ćwiczenia>>

Co chciałem zrobić? Po stworzeniu obiektu `u1` chciałem go skopiować do `u2`, a następnie zmodyfikować `u2`, żeby pokazać, że jest innym, niezależnym obiektem. Jednak napisy umieszczone w liście są identyczne i wskazują, że zarówno `u1`, jak i `u2` mają wartość 3/4.

Co zrobiłem w rzeczywistości? Stworzyłem obiekt `u1`, następnie zadeklarowałem referencję `u2` i zapisałem do niej wartość referencji `u1`. W tej chwili obie referencje wskazują na ten sam obiekt, a więc wywołanie `u2.setLicznik` da identyczny rezultat jak wywołanie `u1.setLicznik`. I stąd taki rezultat[19].

Uwaga! Operator przypisania nie nadaje się do kopiowania obiektów, pozwala jedynie na kopiowanie referencji.

W Javie nie ma możliwości zdefiniowania operatora przypisania dla naszej klasy. W ogóle nie ma możliwości przeciążania operatorów przez programistę. Kopiowanie obiektów powinno odbywać się przez wykorzystanie metody `clone`, którą wszystkie obiekty dziedziczą z klasy `Object`. W przeciwieństwie do metody `equals`, której napisanie wymaga znajomości interpretacji klasy, `clone` może zostać stworzone automatycznie poprzez kompilator. Jej działanie polega wówczas na skopiowaniu wartości wszystkich właściwości kopiowanego obiektu, czyli sklonowaniu obiektu i jego aktualnego stanu. Problem polega na tym, że `clone` jest zadeklarowana w `Object` jako `protected` (z ang. *chroniony*). Oznacza to, że jej zakres dostępności jest taki sam jak właściwości zadeklarowanych jako prywatne, z tą ważną różnicą, że właściwości prywatne są niewidoczne w klasach dziedziczących, a chronione są. Konieczne jest zatem upublicznienie metody `clone`, a więc nadpisanie jej ze zmianą zakresu na publiczny. Idea jest prosta — tworzymy metodę `clone` o identycznej sygnaturze jak metoda obiektu bazowego z jedyną różnicą w deklaracji zakresu dostępności i wywołujemy z niej metodę pierwotną:

```
public Object clone()
  {
     return super.clone();
  }
```

[19] To jest właśnie typowe miejsce, gdzie zaciera się różnica między obiektem i referencją do niego. Oczywiście `u1` jest tylko referencją do obiektu, a nie samym obiektem. Błąd, który popełniłem w powyższym kodzie, bierze się właśnie z zapomnienia o tej różnicy.

Zauważmy, że aby dostać się do metody `clone`, która obecnie jest nadpisana, należy wykorzystać bardzo poręczną referencję `super`. Oznacza ona referencję do naszego obiektu zrzutowaną na klasę bazową, co powoduje, że `super.clone` nie oznacza bieżącej metody, a metodę `clone` klasy `Object`[20].

W rzeczywistości sprawa nieco się komplikuje z dwóch powodów. Po pierwsze, metoda `clone` zadeklarowana w klasie `Object` może zwracać wyjątek `CloneNotSupportedException` (z ang. *klonowanie nie jest możliwe*). Po drugie, klasa, która chce korzystać z metody `clone`, musi zaimplementować interfejs `Cloneable`.

Ćwiczenie 2.34.

Aby napisać metodę pozwalającą na klonowanie obiektów typu `Ulamek`, tj. tworzenie obiektu i inicjowanie jego stanu w taki sposób, aby był identyczny ze wzorem:

1. Przechodzimy do edycji klasy `Ulamek`.
2. Definiujemy w niej następującą metodę `clone`:

```
public Object clone() throws CloneNotSupportedException
  {
    try
    {
      return super.clone();
    }
    catch(CloneNotSupportedException exc)
    {
      throw exc;
    }
  }
```

3. Aby klasa `Ulamek` mogła korzystać z metody `clone` musi implementować interfejs `Cloneable`. Musimy zatem odpowiednio zmodyfikować sygnaturę klasy `Ulamek`:

```
public class Ulamek implements Cloneable
{
  private int licznik, mianownik=1;

  //dalsza część klasy
```

<<koniec ćwiczenia>>

Ćwiczenie 2.35.

Aby przetestować działanie metody `clone` w aplecie `UlamekDemo`:

1. Wracamy do edycji apletu `UlamekDemo`.
2. Modyfikujemy operację kopiowania w metodzie zdarzeniowej napisanej w ćwiczeniu 2.34 w taki sposób, żeby wykorzystać do tego metodę `clone`:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        try
        {
            Ulamek u1 = new Ulamek(1,3);
            //Ulamek u2 = u1;
            Ulamek u2 = (Ulamek)u1.clone();
```

[20] Dostęp do konstruktora klasy bazowej jest również możliwy przez wywołanie metody `super()`, ale tylko z pierwszej linii konstruktorów klasy rozszerzającej.

```
            u2.setLicznik(3);
            u2.setMianownik(4);
            listModel.addElement("u1: "+u1.getLicznik()+"/"+u1.getMianownik());
            listModel.addElement("u2: "+u2.getLicznik()+"/"+u2.getMianownik());
        }
        catch (Exception exc)
        {
            listModel.addElement("Błąd: "+exc.getMessage());
        }
    }
```

3. Kompilujemy i uruchamiamy aplet.
4. Po uruchomieniu apletu uruchamiamy metodę testującą klikając przycisk apletu.

<<koniec ćwiczenia>>

Teraz wynik powinien być zgodny z oczekiwaniami.

Metoda `clone` zgłasza co prawda wyjątek `CloneNotSupportedException`, ale ponieważ obsługujemy już bazowy wyjątek `Exception`, to i ten zostanie automatycznie uwzględniony.

Kiedy trzeba samodzielnie napisać metodę clone?

Metodę `clone` trzeba modyfikować tylko wówczas, gdy właściwościami klasy są referencje do obiektów. Wyobraźmy sobie alternatywną definicję klasy `Ulamek`[21]

```
class UlamekTablica
{
  public int[] lm=new int[2];
}
```

korzystającą do przechowania licznika i mianownika z dwuelementowej tablicy liczb całkowitych typu `int`. Referencja `lm` typu `int[]` jest referencją do tej tablicy (a należy pamiętać, że w Javie tablice są obiektami). Pominęliśmy w klasie metody dostępu i uczyniliśmy referencję `lm` publiczną, żeby uprościć definicję klasy i ułatwić dostrzeżenie istoty problemu. Dodajmy do tej klasy możliwość klonowania w identyczny sposób jak w klasie `Ulamek`:

```
class UlamekTablica implements Cloneable
{
  public int[] lm=new int[2];

  public Object clone() throws CloneNotSupportedException
  {
    try
    {
      return super.clone();
    }
    catch(CloneNotSupportedException exc)
    {
      throw exc;
    }
```

[21] Ponieważ to jest klasa, którą będziemy wykorzystywać tylko w tym podrozdziale, można ją zadeklarować na końcu pliku apletu *UlamekDemo.java*, a potem usunąć.

```
  }
}
```

Dodaliśmy deklaracje wykorzystania interfejsu do sygnatury i metodę `clone` identyczną jak wykorzystana w klasie `Ulamek`.

Teraz zmodyfikujmy metodę zdarzeniową apletu tak, żeby testowała nową klasę:

```
void jButton1ActionPerformed(ActionEvent e)
  {
    UlamekTablica ut1=new UlamekTablica();
    ut1.lm[0]=1;
    ut1.lm[1]=3;
    try
    {
      UlamekTablica ut2 = (UlamekTablica)ut1.clone();
      ut2.lm[0]=3;
      ut2.lm[1]=4;
      list1.add("ut1: "+ut1.lm[0]+"/"+ut1.lm[1]);
      list1.add("ut2: "+ut2.lm[0]+"/"+ut2.lm[1]);
    }
    catch(Exception exc)
    {
      list1.add("Błąd: "+exc.getMessage());
    }
}
```

Odtworzyliśmy działanie poprzedniej wersji, zmieniając jedynie nazwę klas i sposób dostępu do licznika i mianownika. Tworzymy obiekt `ut1` z wartością 1/3, klonujemy go i wartość nowego obiektu zmieniamy na 3/4.

Jaki jest efekt? Otóż otrzymamy dwie identyczne wartości równe 3/4. Dlaczego? Ponieważ metoda `clone` skopiowała dokładnie wszystkie właściwości z obiektu `ut1` i zapisała je do obiektu `ut2`. Wszystkie, czyli w naszym przypadku jedną, a mianowicie referencję do dwuelementowej tablicy `int`. Identyczna wartość tej referencji w obiekcie `ut2` i `ut1` oznacza nic innego niż to, że wskazują na tę samą tablicę. Pomimo że obiekt `u2` jest utworzony i powstaje nowa tablica tworzona podczas tworzenia obiektu-klonu w czasie klonowania, to jednak właściwość `lm` nowego obiektu nie wskazuje na właściwą tablicę, tylko na tablicę z obiektu kopiowanego `u1`.

Jak można to poprawić? Trzeba zmodyfikować metodę `clone` w następujący sposób:

```
public Object clone() throws CloneNotSupportedException
  {
    try
    {
      Object wynik=super.clone();
      int[] nowe_lm=new int[2];
      nowe_lm[0]=this.lm[0];
      nowe_lm[1]=this.lm[1];
      ((UlamekTablica)wynik).lm=nowe_lm;
      return wynik;
    }
    catch(CloneNotSupportedException exc)
    {
      throw exc;
```

```
    }
  }
```

Podczas klonowania tworzymy nową tablicę, kopiujemy do niej zawartość starej i referencję do niej zapisujemy jako wartość właściwości `lm` w sklonowanym obiekcie. Teraz wynik testu będzie już prawidłowy.

Zapamiętaj! Metodę `clone` musisz modyfikować, jeżeli chcesz, aby możliwe było kopiowanie obiektów klas z referencjami jako właściwości.

Obiekty statyczne. Modyfikatory static i final. Dziedziczenie

Po tej dygresji wróćmy do naszej pierwotnej klasy `Ulamek`. Chcielibyśmy dysponować gotowymi obiektami klasy `Ulamek`, które będą odpowiadały najczęściej wykorzystywanym wartościom, podobnie jak np. w wykorzystywanej w poprzednim rozdziale klasie `Color` znajdują się stałe określające predefiniowane kolory.

Ćwiczenie 2.36.

Aby w klasie `Ulamek` zdefiniować obiekty typu `Ulamek` o wartościach odpowiadających zeru i jedności:

1. Przechodzimy do edycji klasy `Ulamek`.
2. Dodajemy do niej deklaracje następujących właściwości:

```
public class Ulamek implements Cloneable
{
  private int licznik, mianownik=1;

  public static final Ulamek zero  = new Ulamek(0);
  public static final Ulamek jeden = new Ulamek(1);

  //dalsza czesc klasy
```

<<koniec ćwiczenia>>

Zadeklarowaliśmy w ten sposób wewnątrz klasy `Ulamek` dwa obiekty typu `Ulamek`. Dziwne, prawda? Nie do pomyślenia w C++. A jednak możliwe w Javie i bardzo wygodne.

Spójrzmy na deklaracje pierwszego obiektu. Gdybyśmy zadeklarowali go jako:

```
public Ulamek zero = new Ulamek(0);
```

tzn. gdybyśmy pominęli modyfikatory `static` i `final`, uzyskalibyśmy bardzo dziwny efekt przy próbie wykorzystania klasy `Ulamek`. Po stworzeniu klasy `Ulamek` wirtualna maszyna Javy próbuje stworzyć klasę `Ulamek` dla obiektu `zero`. W porządku. Ale ponieważ `zero` jest również typu `Ulamek`, w nim też trzeba będzie stworzyć klasę `Ulamek` dla kolejnego obiektu `zero`. I tak w nieskończoność, co spowoduje zapętlenie się apletu.

Do uniknięcia takiej sytuacji służy modyfikator `static`. Ogólnie rzecz biorąc, służy on do tego, żeby kopia obiektu `zero` była robiona tylko raz. I to nie tylko w sytuacji opisanej powyżej, bo każdy obiekt typu `Ulamek` w naszym aplecie będzie zawierał referencje o nazwie `zero`, która odnosi się do jednego, wspólnego dla całej klasy obiektu. Obiekt ten istnieje nawet wówczas, gdy nie istnieje żaden obiekt klasy `Ulamek` stworzony za pomocą operatora `new`. Obiekty takie nazywamy statycznymi lub klasowymi, bo związane są z samą klasą `Ulamek`, a nie tylko z jej obiektami.

Ćwiczenie 2.37.

Aby wykorzystać obiekty statyczne `Ulamek.zero` i `Ulamek.jeden` do inicjacji referencji w testowej metodzie zdarzeniowej:

1. Jeszcze raz przenosimy się do klasy apletu `UlamekDemo`.
2. Korzystanie z obiektów statycznych jest bardzo wygodne (pamiętamy korzystanie takich obiektów w klasie `Color`). Można to sprawdzić inicjując referencje w metodzie zdarzeniowej apletu w nowy sposób:

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        try
        {
            Ulamek zero = Ulamek.zero;
            Ulamek jeden = Ulamek.jeden;
            listModel.addElement("zero:  "+zero.getLicznik()+"/"+zero.getMianownik());
            listModel.addElement("jeden: "+jeden.getLicznik()+"/"+jeden.getMianownik());
        }
        catch (Exception exc)
        {
            listModel.addElement("Błąd: "+exc.getMessage());
        }
    }
```

<<koniec ćwiczenia>>

Zauważmy, że właściwości `zero` i `jeden` pojawią się na liście obiektów dostępnych po napisaniu kropki za nazwą klasy `Ulamek` (*Ctrl+H*)

Modyfikator `static` może również dotyczyć metod, co pokażemy niżej, oraz „zwykłych" pól (typów prostych). Zawsze oznacza dla użytkownika klasy tyle, że ten element klasy jest dostępny zawsze, nawet bez tworzenia obiektu.

Drugi z nowych modyfikatorów przy deklaracji obiektów statycznych to `final`. Oznacza, że w trakcie dziedziczenia definicja obiektów `zero` i `jeden` nie może się już zmienić. Modyfikator `final` może również dotyczyć „zwykłych" (tj. niestatycznych) pól i metod.

Ćwiczenie 2.38.

Aby zdefiniować kolejne obiekty statyczne `pol` i `cwierc` mające wartości 1/2 i 1/4:

1. Spróbujmy następującej deklaracji:

```
public static final Ulamek polowa=new Ulamek(1,2);
```

2. Po niepowodzeniu poprzedniej próby dodajmy na końcu pliku *Ulamek.java* definicję klasy:

```
class UlamekNoException extends Ulamek
{
  public UlamekNoException(int licznik,int mianownik)
  {
    try
    {
      setMianownik(mianownik);
      setLicznik(licznik);
    }
    catch(Exception exc)
    {
      //tu nie ma nic, wiec odpowiedzialność za poprawną
      //wartość mianownika spada na użytkownika tej klasy pomocniczej
    }
  }
}
```

3. Teraz możemy do klasy `Ulamek` dodać kolejne definicje obiektów statycznych:

```
public static final Ulamek polowa=new UlamekNoException(1,2);
```

```
public static final Ulamek cwierc=new UlamekNoException(1,4);
```

<<koniec ćwiczenia>>

Niestety, przy próbie zdefiniowania tych zdecydowanie bardziej pożytecznych obiektów statycznych w najprostszy sposób (punkt 1.) uzyskamy komunikat błędu. Dlaczego? Ponieważ konstruktor dwuargumentowy wymaga obsługi wyjątku, a w tym kontekście nie jest to możliwe. Można jednak ten problem ominąć, jednak wymaga to od nas napisania klasy pomocniczej rozszerzającej klasę `Ulamek` (punkt 2.). Będzie to klasa prywatna, tylko na użytek zdefiniowania obiektów statycznych, w której zmodyfikujemy konstruktor w taki sposób, żeby nie zgłaszał wyjątków. Z tego powodu nie powinniśmy jej udostępniać.

Jak widać z kodu znajdującego się w punkcie 2. ćwiczenia jedynym zadaniem klasy `UlamekNoException` jest zastąpienie dwuargumentowego konstruktora klasy bazowej zgłaszającego wyjątek na taki, który tego nie robi. Wszelkie wyjątki zgłaszane przez metodę `setMianownik` są przechwytywane, ale informacja o ich wystąpieniu nie jest przekazywana.

Warunkiem powodzenia powyższego triku jest obecność domyślnego (bezargumentowego) konstruktora w klasie `Ulamek`.

Wreszcie w punkcie 3. definiujemy żądane obiekty statyczne w taki sposób, że referencje do nich są typu `Ulamek`, a więc użytkownik klasy nie musi nawet wiedzieć o użytej przez nas sztuczce.

Niepostrzeżenie nauczyliśmy się również dziedziczenia klas[22]. Jak widać, realizuje się je poprzez dodanie do deklaracji nowej klasy słowa kluczowego `extends` i nazwy klasy bazowej. Klasa, która rozszerza klasę bazową, przejmuje wszystkie jej metody zadeklarowane jako publiczne i chronione. Niedostępne stają się jedynie właściwości zadeklarowane jako prywatne. W takim przypadku niemożliwy jest dostęp z klasy `UlamekNoException` do prywatnych pól `licznik` i `mianownik`. Można jednak użyć metod `setLicznik` i `getLicznik`, bo zadeklarowane zostały jako publiczne.

Oczywiście to nie wyczerpuje zagadnienia dziedziczenia klas, ale na potrzeby tej książki jest to wiedza w zupełności wystarczająca.

Metody statyczne

Zadeklarujmy wewnątrz klasy `Ulamek` metodę statyczną `odwroc`, która będzie zwracać `Ulamek` o zamienionym liczniku i mianowniku względem ułamka podanego w jej argumencie. Gdyby nie obsługa wyjątków zgłaszanych przez konstruktor, jej postać byłaby bardzo prosta:

```
public static Ulamek odwroc(Ulamek ulamek)
{
  return new Ulamek(ulamek.getMianownik(),ulamek.getLicznik());
}
```

W naszej sytuacji musimy metodę uzupełnić o konstrukcję `try...catch`.

Ćwiczenie 2.39.

Aby przygotować statyczną metodę wykonującą operację odwrócenia ułamka podanego w jej argumencie:

Deklarujemy wewnątrz klasy `Ulamek` metodę statyczną `odwroc` o następującej definicji:

```
public static Ulamek odwroc(Ulamek ulamek) throws Exception
  {
    try
    {
      return new Ulamek(ulamek.getMianownik(),ulamek.getLicznik());
    }
    catch(Exception exc)
```

[22] W Javie używa się również terminu *rozszerzanie klas*.

```
        {
          throw exc;
        }
      }
```

<<koniec ćwiczenia>>

Metoda zgłosi wyjątek, gdy spróbujemy odwrócić ułamek z zerem w liczniku.

Ćwiczenie 2.40.

Aby przetestować działanie metody statycznej `odwroc` w testowej metodzie apletu `UlamekDemo`:

1. Przechodzimy do metody zdarzeniowej w aplecie `UlamekDemo` testującej naszą klasę.
2. Sprawdzamy działanie metody `Ulamek.odwroc` korzystając z następującej postaci metody zdarzeniowej:

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        try
        {
            Ulamek u1 = Ulamek.polowa;
            Ulamek u2 = Ulamek.odwroc(u1);
            listModel.addElement("u1: "+u1.getLicznik()+"/"+u1.getMianownik());
            listModel.addElement("u2: "+u2.getLicznik()+"/"+u2.getMianownik());
        }
        catch (Exception exc)
        {
            listModel.addElement("Błąd: "+exc.getMessage());
        }
    }
```

<<koniec ćwiczenia>>

Z metody statycznej nie ma dostępu do niestatycznych pól klasy. To bardzo ważne. Nie ma do nich dostępu, bo nie ma gwarancji, że wywołanie tej metody odbędzie się na rzecz jakiegoś obiektu. Może również zostać wywołana bez niego, tak jak w powyższym kodzie. Mamy jedynie dostęp do obiektów `zero`, `jeden`, `pol` i `cwierc`, ponieważ one są również statyczne. Dlatego nie można za pomocą tak zdefiniowanej statycznej metody obrócić bieżącego obiektu `Ulamek`, a jedynie obiekt podawany przez argument.

Zapamiętaj**!** Z metod statycznych dostępne są jedynie pola i metody statyczne.

Zawsze można oczywiście zastosować następującą konstrukcję:

```
u1 = u1.odwroc(u1);
```

równoważną

```
u1 = Ulamek.odwroc(u1);
```

ale znaczniej wygodniej będzie zdefiniować specjalną metodę niestatyczną. Nic nie stoi na przeszkodzie, aby ta nowa metoda miała identyczną nazwę jak metoda statyczna, jeżeli różni się rodzajem lub ilością argumentów.

Ćwiczenie 2.41.

Aby zdefiniować niestatyczną metodę `odwroc` zamieniającą wartościami pola licznik i mianownik z bieżącego obiektu:

1. Wracamy do klasy `Ulamek`.
2. Nową metodę `odwroc` definiujemy następująco (nie kasując metody statycznej):

```
  public Ulamek odwroc() throws Exception
```

```
{
  try
  {
    return odwroc(this);
  }
  catch(Exception exc)
  {
    throw exc;
  }
}
```

<<koniec ćwiczeń>>

Wykorzystujemy w niej metodę statyczną, żeby nie powtarzać kodu i ułatwić sobie ewentualne zmiany.

Na podobnych zasadach możemy zrealizować inne operacje jednoargumentowe na ułamkach (np. zmiana znaku, podniesienie do kwadratu itp.) oraz operacje dwuargumentowe. Weźmy na przykład dodawanie dwóch ułamków. Zgodnie z elementarnymi wzorami algebry powinno ono odbyć się według wzoru:

$$\frac{a}{b}+\frac{c}{d}=\frac{ad+cb}{bd}.$$

Ponownie napiszemy najpierw metodę statyczną dwuargumentową, a następnie wykorzystamy ją do napisania zwykłej metody jednoargumentowej.

Ćwiczenie 2.42.

Aby w klasie `Ulamek` zdefiniować metodę statyczną o nazwie `dodaj` przyjmującą przez głowę dwa obiekty klasy `Ulamek` i zwracającą ich sumę oraz metodę niestatyczną korzystającą z metody statycznej dodającą podany w argumencie ułamek do bieżącego obiektu:

1. Do klasy `Ulamek` dopisujemy dwuargumentową metodę statyczną `dodaj`:

```
public static Ulamek dodaj(Ulamek u1,Ulamek u2)
  {
    int a=u1.getLicznik();
    int b=u1.getMianownik();
    int c=u2.getLicznik();
    int d=u2.getMianownik();
    //mozemy miec wewnosc, ze b*d jest rozne od zera
    //wiec stosujemy konstruktor bez wyjatkow
    return new UlamekNoException(a*d+b*c,b*d);
  }
```

2. ... oraz jednoargumentową metodę niestatyczną o tej samej nazwie:

```
public Ulamek dodaj(Ulamek u2)
{
  return dodaj(this,u2);
}
```

<<koniec ćwiczenia>>

Znowu stosujemy konstruktor z klasy `UlamekNoException`, bo dodawanie jest bezpieczne — mianownik tworzymy z pomnożenia dwóch liczb, które na pewno są różne od zera. Podobnie będzie w odejmowaniu i mnożeniu. Jedynie metoda `podziel` powinna móc zgłaszać wyjątki.

Rzutowanie na łańcuch

Sposób prezentowania ułamka jako łańcucha w liście, w którym musimy korzystać z konstrukcji typu `u1.getLicznik()+"/"+u1.getMianownik()` nie jest zbyt wygodny. Dodajmy zatem do klasy `Ulamek` bezargumentową metodę `toString`, która będzie tworzyła łańcuch odpowiadający wartości pól `licznik` i `mianownik`.

Nic prostszego. Musimy po prostu umieścić w tej metodzie kod, który zazwyczaj wykorzystywaliśmy do zaprezentowania wartości ułamka w metodzie testowej:

Ćwiczenie 2.43.

Aby w klasie `Ulamek` zdefiniować metodę `toString` konwertującą bieżący obiekt do łańcucha i zastosować ją w metodzie zdarzeniowej apletu `UlamekDemo`:

1. W klasie `Ulamek` umieszczamy definicję metody `toString`:

```
public String toString()
    {
    return String.valueOf(licznik)+"/"+String.valueOf(mianownik);
    }
```

2. W aplecie `UlamekDemo` modyfikujemy metodę testową w następujący sposób:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        listModel.addElement("polowa: "+Ulamek.polowa.toString());
}
```

<<koniec ćwiczenia>>

Ale to nie wszystko. Metoda `toString` nie jest bowiem taka zupełnie „zwykła”. Zauważmy, że tak naprawdę nadpisaliśmy metodę z klasy `Object`. Okazuje się, że podobnie jak metody `clone` i `equals` z tej klasy, także `toString` pełni specjalną rolę. Otóż jest to metoda wykorzystywana przez operator `+`, jeżeli z lewej strony stoi łańcuch, a z prawej obiekt klasy `Ulamek`. Oznacza to, że możemy powyższą metodę uprościć do następującej postaci:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        listModel.addElement("polowa: "+Ulamek.polowa);
}
```

To jedyny przypadek, kiedy w Javie mamy wpływ na działanie operatora. Możemy nawet powiedzieć, że mamy w ten sposób możliwość jego przeciążenia. W innych sytuacjach operatory są zasadniczo w tym języku nie do ruszenia.

Rzutowanie na liczbę double

W podobny sposób jak `toString` możemy zdefiniować metodę `toDouble`. Tym razem nie będzie ona miała żadnego drugiego dna.

Ćwiczenie 2.44.

Aby zdefiniować w klasie `Ulamek` metodę `toDouble` zwracającą liczbę typu `double` o wartości odpowiadającej bieżącemu stanowi obiektu:

W klasie `Ulamek` umieszczamy następującą metodę:

```
public double toDouble()
  {
    return licznik/(double)mianownik;
  }
```

<<koniec ćwiczenia>>

Gdybyśmy nie zrobili rzutowania w liczniku lub mianowniku, otrzymalibyśmy zły wynik, bowiem operator dwuargumentowy `/` jest tak przeciążony, że gdy jego oba argumenty (tj. liczba z lewej i prawej strony) są typu `int`, to wynik jest również zaokrąglany do `int`. Zupełnie identycznie jak w C++.

Ćwiczenie 2.45.

Aby nową metodę przetestować w metodzie zdarzeniowej `UlamekDemo`:

W aplecie `UlamekDemo` modyfikujemy metodę testującą wg wzoru:

```
void button1_actionPerformed(ActionEvent e)
  {
    list1.add("polowa: "+Ulamek.polowa.toDouble());
  }
```

<<koniec ćwiczenia>>

Metoda upraszczająca ułamek

Na koniec jako przykład zastosowania poleceń sterowania przepływem oraz operatorów arytmetycznych przedstawiam bez komentarza definicję metody upraszczającej ułamek, którą można dodać do definicji klasy:

```
public void uprosc()
  {
    //NWD
    int mniejsza=(Math.abs(licznik)<Math.abs(mianownik))
        ?Math.abs(licznik):Math.abs(mianownik);
    for(int i=mniejsza;i>0;i--)
      if ((licznik%i==0) && (mianownik%i==0))
      {
        licznik/=i;
        mianownik/=i;
      }
    //znaki
    if (licznik*mianownik<0)
    {
      licznik=-Math.abs(licznik);
      mianownik=Math.abs(mianownik);
    }
    else
    {
      licznik=Math.abs(licznik);
      mianownik=Math.abs(mianownik);
    }
  }
```

*

Drogi Czytelniku, jeżeli pierwszy raz w życiu spotkałeś się z pojęciem klasy, to zatrzymaj się tutaj i wykonaj dodatkowe ćwiczenia. Powtórz wszystkie kroki z tego podrozdziału budując np. klasę `Complex` opisującą liczby zespolone lub klasę `RownanieKwadratowe`, która na podstawie trzech współczynników równania kwadratowego znajduje jego pierwiastki. Zapewniam, że zrozumienie, czym są klasy, i docenienie zalet programowania zorientowanego obiektowo jest bardzo ważne nie tylko w Javie, ale i w każdym innym współczesnym języku programowania.

Kilka dodatkowych wiadomości o klasach w skrócie

Klasa abstrakcyjna

Klasa może być abstrakcyjna, to znaczy, że nie jest dozwolone stworzenie obiektu tej klasy. Zazwyczaj klasy abstrakcyjne mają metody abstrakcyjne, które są zadeklarowane, ale nie są zdefiniowane.

Jaki jest powód tworzenia klas abstrakcyjnych?

Klasy abstrakcyjne mogą być po prostu „półproduktem", klasą, która przygotowuje podstawowe właściwości i jest punktem wyjścia do dalszych możliwych rozwinięć. Przykładem mogłaby być klasa implementująca „dwójkę", tj. parę liczb z możliwością ich odczytu i zapisu. Chcielibyśmy móc wykonywać operacje arytmetyczne na takiej dwójce (dodawanie, odejmowanie itp.). Deklarujemy zatem metody abstrakcyjne `dodaj`, `pomnoz`. Jednak możemy je zaimplementować dopiero wtedy, gdy będziemy znali interpretację tej dwójki. Inna będzie metoda `dodaj` dla dwójki implementującej ułamek, inna dla punktu na płaszczyźnie, a inna dla dwójki implementującej liczbę zespoloną. Klasa abstrakcyjna jest w tym przypadku sposobem, żeby nie powtarzać kodu służącego do modyfikacji i odczytania pól. Może również dostarczać podstawowe formy konstruktorów inicjujących właściwości, a jednocześnie wyznacza, jakie metody powinny się znaleźć w klasach rozszerzonych.

Inny typowy powód to dostarczanie użytkownikowi jakiegoś mechanizmu. Przyjmijmy, że tworzymy klasę służącą do tworzenia wykresów funkcji `Plot`. Klasa przygotowuje układ współrzędnych, opis osi itd. W klasie zadeklarowana jest metoda abstrakcyjna `funkcja`, do której odnoszą się metody przygotowujące wykres. Ale o tym, jaką funkcję pokazać, decyduje użytkownik, więc musi on rozszerzyć klasę `Plot` i zdefiniować metodę `funkcja`. Poza tym wszystko inne jest już gotowe.

My pokażemy znacznie prostszy przykład. Dodajmy na końcu pliku naszego testowego apletu klasę zadeklarowaną w następujący sposób:

```
abstract class KlasaAbstrakcyjna
{
  public String opis="Klasa abstrakcyjna";
  abstract String przedstawSie();
}
```

Klasa jest abstrakcyjna — świadczy o tym modyfikator umieszczony na początku jej sygnatury. Posiada również deklarację metody abstrakcyjnej `przedstawSie`[23]. Nie można stworzyć obiektu klasy `KlasaAbstrakcyjna`.

Gdzie można definiować i rozszerzać klasy?

Odpowiedź brzmi: niemal wszędzie. Dowolność miejsc, w których możemy definiować i rozszerzać klasy w Javie, jest zaskakująca. Wyżej tworzyliśmy już klasy poza klasą apletu czy w osobnym pliku, ale do wyczerpania możliwości jest jeszcze daleko. Spójrzmy na kilka przykładów (klasę apletu przedstawiam schematycznie pomijając wszystkie jej właściwości i metody poza naszą metodą zdarzeniową służącą do testowania klasy `Ulamek`.)

W aplecie `UlamekDemo` umieścimy definicje klas rozszerzających klasę `KlasaAbstrakcyjna` z poprzedniego podrozdziału. W każdej z nich definiujemy tylko metodę `przedstawSie`[24]. To, że w tym przykładzie rozszerzamy akurat klasę abstrakcyjną, nie ma znaczenia dla miejsca, w którym ją definiujemy.

Listing 2.7. Schemat klasy apletu z definicjami klasy zewnętrznej, wewnętrznej, lokalnej i anonimowej

```
public class UlamekDemo extends javax.swing.JFrame
{
  KlasaAbstrakcyjna kz=new KlasaZewnetrzna();
  KlasaAbstrakcyjna kw=new KlasaWewnetrzna();
```

[23] Klasa abstrakcyjna może również posiadać inne nieabstrakcyjne metody i właściwości, ale pominęliśmy je tutaj dla uproszczenia obrazu.

[24] Jest to metoda abstrakcyjna klasy abstrakcyjnej, którą rozszerzamy, więc nie tylko możemy, ale musimy ją zdefiniować jeżeli klasa rozszerzona nie ma być także abstrakcyjna.

```
  private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

    class KlasaLokalna extends KlasaAbstrakcyjna
    {
      public String przedstawSie(){return "Klasa lokalna";}
    }

    KlasaAbstrakcyjna kl=new KlasaLokalna();
    KlasaAbstrakcyjna ka=new KlasaAbstrakcyjna()
                      {
                        public String przedstawSie(){return "Klasa anonimowa";}
                      };
    listModel.addElement(kz.przedstawSie());
    listModel.addElement(kw.przedstawSie());
    listModel.addElement(kl.przedstawSie());
    listModel.addElement(ka.przedstawSie());
  }

class KlasaWewnetrzna extends KlasaAbstrakcyjna
{
  public String przedstawSie(){return "Klasa wewnętrzna";}
}
}

abstract class KlasaAbstrakcyjna
{
  abstract String przedstawSie();
}

class KlasaZewnetrzna extends KlasaAbstrakcyjna
{
  public String przedstawSie(){return "Klasa zewnętrzna";}
}
```

Oczywiście już wiemy, że definicję klasy możemy umieścić poza klasą, ale w tym samym pliku. Dla klasy apletu (`UlamekDemo`) będzie to klasa zewnętrzna. Ponieważ nie może być klasą publiczną (to miejsce zajmuje już klasa apletu), jest widoczna tylko wewnątrz pakietu. Klasa zadeklarowana wewnątrz klasy (klasa wewnętrzna) jest widoczna tylko wewnątrz tej klasy. Można również zdefiniować klasę lokalną wewnątrz metody. Warto to robić, gdy mamy pewność, że z klasy nie będziemy korzystać nigdzie indziej niż w tej jednej metodzie. Najbardziej „zlokalizowana” jest jednak klasa anonimowa. Definiujemy ją w momencie tworzenia obiektu i tylko dla tego jednego obiektu. Postępujemy w tym przypadku tak, jak gdybyśmy tworzyli obiekt klasy bazowej, ale po argumentach konstruktora dołączamy definicję dodatkowych właściwości lub metod. W naszym przypadku definicję metody `przedstawSie`.

Obiekty klasy zewnętrznej i wewnętrznej mogą być stworzone jako właściwości klasy — są one bowiem znane w momencie tworzenia obiektu. Obiekt klasy lokalnej może powstać jedynie w metodzie, w której jest zadeklarowana klasa, natomiast polecenie utworzenia klasy anonimowej w takiej postaci, jak w powyższym kodzie, może być umieszczone zarówno w metodzie, jak i w obrębie definicji klasy.

O wyborze miejsca deklaracji klasy może decydować jeszcze jeden ważny czynnik. Klasy zadeklarowane wewnątrz innej klasy (w naszym przypadku klasy `KlasaWewnetrzna`, `KlasaLokalna` i klasa anonimowa) widzą publiczne i prywatne metody klasy, w której zostały umieszczone (w powyższym przykładzie jest to klasa

UlamekDemo). Widzą również te właściwości, które zadeklarowane są jako final. To pozwala na tworzenie klas-dzieci, które mogą modyfikować stan klasy-matki i realizować część jej zadań. Typowy przykład takiego zastosowania klas wewnętrznych, lokalnych lub anonimowych to tworzenie klasy wątku lub tworzenie klas nasłuchiwaczy jako klas umieszczonych wewnątrz klasy macierzystej.

Dynamiczne rozpoznawanie klasy obiektu

Rozpoznawanie typów w trakcie działania programu to złożone zagadnienie. My potrzebujemy tylko najbardziej podstawowych rzeczy, a mianowicie rozpoznawania klasy obiektu, który jest przypisany do referencji, być może innego typu niż właściwa dla niego klasa.

Typowy przykład mieliśmy w aplecie DwieKostki, a potem w metodzie equals klasy Ulamek. W aplecie DwieKostki musieliśmy korzystać z operatora rozpoznawania typu ze względu na to, że użyta tam metoda getComponent zwraca referencję do klasy Component. Nie wiemy z góry, które z pobieranych w ten sposób referencji reprezentują suwaki Scrollbar, a które są innymi komponentami. Bardzo podobnie wyglądał problem w przypadku metody equals klasy Ulamek.

W obu przypadkach korzystaliśmy z dwuargumentowego operatora instanceof

```
obiekt instanceof Klasa
```

który zwraca wartość prawdziwą, jeżeli *obiekt* jest typu *Klasa* lub może być do niego zrzutowany bez wywołania wyjątku ClassCastException.

Więcej informacji o typie obiektu można uzyskać korzystając z klasy Class przechowującej pełną informację o klasie (pełna nazwa, dostępne konstruktory, metody, pola itd.). Obiekt tej klasy można pobrać dla każdej klasy i obiektu. Jedną z metod tej klasy jest isInstance przyjmująca jako argument obiekt, a zwracająca prawdę, jeżeli obiekt jest typu klasy, na rzecz której wykonana została metoda:

```
Klasa.class.isInstance(obiekt)
```

W przypadku apletu DwieKostki warunek sprawdzający, czy referencja komponent odnosi się do obiektu typu JScrollbar, wyglądałby następująco (**zakładam, że odpowiednie klasy są zaimportowane**):

```
for(int i=0; i<this.getComponentCount(); i++)
{
  Component komponent = this.getComponent(i);
  //if (komponent instanceof Scrollbar)
  if (JScrollbar.class.isInstance(komponent))
    {
      JScrollbar suwak = (JScrollbar) komponent;
      suwak.setMaximum(max);
      suwak.setVisibleAmount(max/20);
    }
}
```

Ostatnia możliwość to sprawdzenie nazwy klasy obiektu:

```
komponent.getClass().getName() == "javax.swing.JScrollbar"
```

Tym razem, korzystając z metody getClass zdefiniowanej już w klasie bazowej java.lang.Object, pobieramy obiekt klasy Class dla istniejącego obiektu. Ten sam, który można również otrzymać od klasy korzystając z konstrukcji *Klasa*.class. Następnie na rzecz obiektu klasy Class wywołujemy metodę getName, która zwraca nazwę klasy razem z nazwą pakietu.

To tylko mały skrawek możliwości rozpoznawania klas obiektów w trakcie działania aplikacji, przycięty znacznie na potrzeby ćwiczeń umieszczonych w tej książce.

Kilka słów o interfejsach

Nie chodzi tym razem o graficzny interfejs apletu lub aplikacji, ale o strukturę języka Java podobną do klasy abstrakcyjnej. Konkretny interfejs reprezentuje pewną właściwość lub zdolność obiektu. Zdefiniujmy np.

interfejs `Grzeczny`. Klasa, która wykorzystuje ten interfejs, będzie grzeczna, tzn. będzie potrafiła się ładnie przedstawić. Zatem interfejs `Grzeczny` oznacza, że klasa, która go wykorzystuje, ma metodę służącą do przedstawiania się. Widać podobieństwo do klasy abstrakcyjnej, którą zdefiniowaliśmy wyżej.

Zadeklarujmy zatem interfejs `Grzeczny` i grzeczną klasę:

```
interface Grzeczny
{
  String przedstawSie();
}

class GrzecznaKlasa implements Grzeczny
{
  public String przedstawSie(){return "Grzeczna klasa";}
}
```

Deklaracja interfejsu różni się od klasy użytym słowem kluczowym `interface`. W przeciwieństwie do klasy abstrakcyjnej interfejs może zawierać jedynie deklaracje metod bez ich definicji. Modyfikator `abstract` nie jest w tym przypadku konieczny.

Po co są interfejsy, jeżeli w Javie są klasy abstrakcyjne? Otóż dlatego, że w tym języku nie ma możliwości dziedziczenia z wielu klas bazowych jednocześnie. Interfejsy są rozwiązaniem alternatywnym — można w jednej klasie wykorzystać dowolną ilość interfejsów. Oczywiście metody zadeklarowane w nich wszystkich muszą wówczas być zdefiniowane.

Interfejsy poznamy znacznie lepiej na przykładzie klas nasłuchujących, których mechanizm zostanie wyjaśniony w rozdziałach 4. i 6.