W. WIECZOREK, M. BORYCZKA, R. SIMIŃSKI

PROGRAMOWANIE OBIEKTOWE – ASPEKTY IMPLEMENTACYJNE

INSTYTUT INFORMATYKI UNIWERSYTETU ŚLĄSKIEGO

Spis treści

C++ 7

Java 75

Haskell 119

Python 139

Bibliografia 165

Przygotowanie tego skryptu zostało w całości sfinansowane ze środków projektu „Informatyka inżynierska — Kierunek Zamawiany Uniwersytetu Śląskiego w Katowicach", w ramach Programu Operacyjnego Kapitał Ludzki, współfinansowanego ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

C++

Programowanie obiektowe — wprowadzenie

Rozdział ten poświęcony jest prezentacji podstawowych koncepcji programowania obiektowego w języku C++, kolejne podrozdziały obejmują omówienie niezbędnych podstaw teoretycznych ilustrowanych przykładami oraz zestaw ćwiczeń do samodzielnego wykonania. Zakłada się, że czytelnik zna podstawy programowania nieobiektowego i potrafi tworzyć proste programy tego typu wykorzystując notację języka C++.

Język C++ oferuje wiele różnorodnych mechanizmów programowania obiektowego. Nawet pobieżna ich prezentacja przekracza ramy tego opracowania, dlatego zdecydowano się na dokładnie omówienie wybranych zagadnień o charakterze fundamentalnym, wierząc, że na tej dobrej podbudowie indywidualne poznanie dalszych elementów programowania w C++ będzie łatwiejsze.

Obiektowość w analizie, projektowaniu i programowaniu

W podrozdziale tym dokonamy syntezy programu, który będzie rozwiązywać prosty problem — obliczania pola kwadratu. Jego prostota jest zabiegiem świadomym, celem naszym jest bowiem przystępne przedstawienie obiektowych mechanizmów języka C++. Rozpoczniemy od przedstawienia rozwiązania nieobiektowego a następnie przedstawimy koncepcję rozwiązania obiektowego oraz przykładową implementację w języku C++.

Obliczanie pola kwadratu — analiza problemu

Rozważmy następujące zadanie:

Przykład 1: Należy napisać program umożliwiający wyliczenie pola kwadratu na podstawie podanej przez użytkownika długości boku. Program kontroluje poprawność liczby będącej dana wejściową — długość boku musi być liczba nieujemną.

Przykładowe, nieobiektowe rozwiązanie może być następujące:

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main() {
```

```
  double bok;
  cout << "\nObliczam pole kwadratu" << endl;
  do {
    cout << "\nDlugosc boku: " << flush;
    cin >> bok;
    if( bok < 0 )
      cout << "Dlugosc boku musi byc nieujemna.\n";
  } while( bok < 0 );
  cout << "Pole kwadratu : " << bok * bok << endl;
  return EXIT_SUCCESS;
}
```

Wyznaczenie pola kwadratu dla znanej długości boku — zawartej przykładowo w zmiennej `dlBoku` — jest oczywiste i natychmiastowe: podnosimy wartość zmiennej `dlBoku` do kwadratu i otrzymujemy rozwiązanie. Zapomnijmy jednak na chwilę, że rozwiązanie jest tak proste, więcej poudawajmy przez chwilę, że nie mamy w tym momencie tzw. „zielonego pojęcia" jak wyznaczyć pole kwadratu.

Taka sytuacja jest dość typowa w momencie rozpoczęcia prac nad nowym programem — należy dokonać analizy problemu, jej celem jest znalezienie metod oraz środków jego rozwiązania.

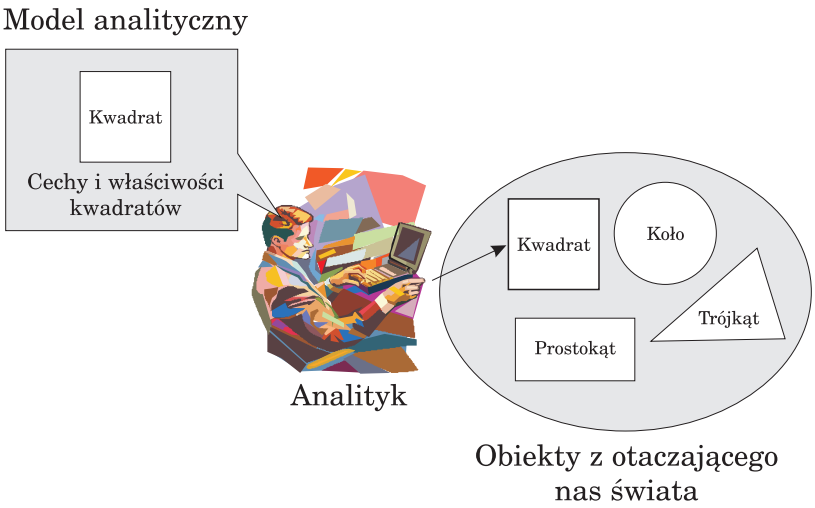
Obliczanie pola kwadratu — analiza obiektowa

W przypadku obiektowego podejścia do projektowania i programowania możemy stwierdzić, że:

> Celem *analizy obiektowej* jest stworzenie modelu docelowego systemu, w którym każdy element świata rzeczywistego odwzorowany jest odpowiednim obiektem (lub ich grupą), należącym do określonej klasy. Interakcja pomiędzy obiektami jest odbiciem procesów zachodzących w rzeczywistości.

Jakie obiekty występują w problemie, który mamy rozwiązać? Rozważania dotyczą czegoś, co moglibyśmy nazwać roboczo *światem figur* — figur płaskich, dla których nie ważne jest zakotwiczenie w danym miejscu płaszczyzny a jedynie ich geometryczne parametry: pole, obwód. Kwadrat, jest rzeczywiście taką figur i dla naszego problemu istotne jest to, co służy do wyznaczenia jego pola (no i może też — patrząc przyszłościowo — obwodu). Zatem celem analizy obiektowej w naszym przypadku jest wyodrębnienie najistotniejszych cech kwadratu — ale nie jakiegoś jednego, szczególnego, ale cech wspólnych dla każdego, możliwego kwadratu, na który możemy trafić rozwiązując nasz problem, ilustruje to rys. 1.

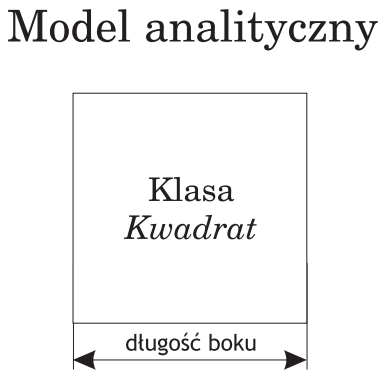
W wyniku analizy cech wszystkich potencjalnych kwadratów, pojawia się pewien wspólny opis ich właściwości. Taki nazwany opis wspólnych właściwości obiektów w modelowaniu obiektowym nazywany jest *klasą*. Każda klasa posiada swoją nazwę i określa właściwości, które będzie posiadał *każdy* obiekt należący do danej klasy. Właściwości, które opisuje klasa są dostosowane do specyfiki problemu — zatem nie dotyczą wszystkich możliwych cech obiektów, lecz



Rysunek 1: Modelowanie obiektowe na przykładzie świata figur

tylko tych, które w danym problemie są istotne. Każdy potencjalny kwadrat, dla którego chcemy policzyć pole, będzie należał do klasy `Kwadrat`.

Jakie właściwości kwadratów interesują nas w rozważanym problemie? Przypomnijmy, że interesuje nas obliczenie pola kwadratu — dowolnego kwadratu. Co jest do tego potrzebne? Wzór na obliczenie pola P kwadratu znamy, lub z łatwością odnajdziemy w podręcznikach: $P = a^2$ lub $P = a * a$, gdzie a to długość boku kwadratu. Zatem najistotniejszą cechą każdego kwadratu będzie długość jego boku. To w tym momencie niezbędne minimum dla realizacji naszego celu. Ale jednocześnie wystarczające do — potencjalnie możliwego w przyszłości — obliczania obwodu. W podobny sposób moglibyśmy uogólnić właściwości innych figur przedstawionych na rys. 1. Nie interesuje nas w tym momencie nic innego — np. nieważny jest kolor kwadratu, jego położenie na płaszczyźnie — to nie są właściwości istotne dla rozważanego problemu. Rys. 2 ilustruje właściwości klasy `Kwadrat` na etapie analizy — możemy szumnie powiedzieć, że zawiera on *model analityczny* naszej klasy opisu kwadratu.



Rysunek 2: Model analityczny — klasa *Kwadrat*

Obliczanie pola kwadratu — projektowanie obiektowe

Od fazy analizy problemu czas przejść do fazy projektowania systemu. Na tym etapie zidentyfikowane obiekty i ich klasy powinny zostać opisane zgodnie z odpowiednimi metodami projektowania obiektowego.

> Celem *projektowania obiektowego* jest opisanie struktury projektowanego systemu w kategoriach obiektów i klas do których należą (zidentyfikowanych w fazie analizy), stosując odpowiednie notacje, odwzorowujące logiczne i fizyczne właściwości obiektów, związki jakie zachodzą pomiędzy klasami, obiektami oraz opisujące ich właściwości statyczne i dynamiczne.

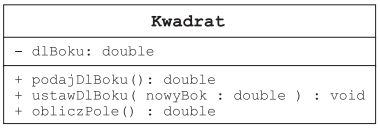
Aktualnie standardem modelowania obiektowego stał się język `UML`. Rys. 3 przedstawia naszą klasę właśnie jako element diagra-

mu klas UML. W projektowane klasie wyodrębniliśmy pole `dlBoku` (skrót *długość boku*), służące oczywiście do przechowywania informacji o długości boku kwadratu, jest to pole typu liczba rzeczywista podwójnej precyzji (typ `double`). Pole otrzymało na diagramie prefiksowy symbol '–', co oznacza, że to pole jest *prywatne* i odwoływać się do niego będzie mógł tylko sam obiekt klasy `Kwadrat`.

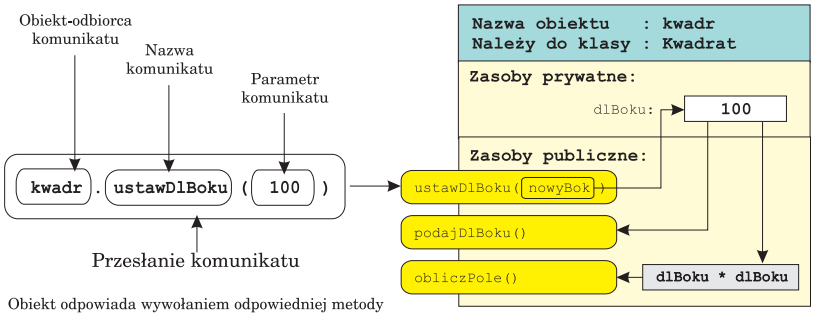
Aby ktokolwiek mógł z otoczenia jakiegoś obiektu klasy `Kwadrat` mógł ustawiać lub pobierać wartość tego pola, należy wbudować w klasę *metody dostępowe*. Jeżeli będziemy chcieli dowiedzieć się jaka jest aktualna długość boku pewnego obiektu, musimy ten obiekt o to „poprosić" wywołaniem metody `podajDlBoku`. Jeżeli będziemy chcieli zmienić długość boku jakiegoś obiektu, będziemy musieli „poprosić" o to obiekt wywołaniem metody `ustawDlBoku`, która w postaci parametru otrzyma liczbę, określającą nową długość boku dla kwadratu. Metody dostępowe `podajDlBoku` oraz `ustawDlBoku` mają charakter administracyjny, nie realizują właściwego przetwarzania danych właściwych dla obiektu, do tego służyć będzie metoda `obliczPole`, będąca funkcją, zatem jej rezultat będzie określał pole kwadratu o długości zapisanej w prywatnym polu `dlBoku`. Wszystkie te metody otrzymały prefiksowy symbol '+', co oznacza, że są metodami publicznymi — ogólnie dostępnymi.

Publiczne metody klasy określają jej *interfejs* — komunikacja z obiektem takiej klasy odbywa się za pośrednictwem nazw metod, co ilustruje rys. 4.

Model projektowy



Rysunek 3: Model projektowy — klasa *Kwadrat*



Rysunek 4: Obiekt klasy *Kwadrat*

Obliczanie pola kwadratu — programowanie obiektowe

Jak wykorzystać zaprojektowaną klasę do realizacji programu w języku C++? Aby można było wykorzystać zaprojektowaną klasę, potrzebny jest przynajmniej jeden jej *obiekt* (inaczej mówi się: *egzemplarz klasy*). Załóżmy na chwilę, że mamy jeden obiekt klasy `Kwadrat`, i że ten konkretny obiekt posiada nazwę `kwadr`. Powołanie do życia obiektu w języku C++ polega na zdefiniowaniu *zmiennej*, będącej *obiektem*, której *typem* jest *nazwa klasy*:

```
Kwadrat kwadr;
```

Z tak zdefiniowanym obiektem możemy się komunikować, przesyłając komunikaty odpowiadające metodom zdefiniowanym w czę-

ści publicznej klasy. Ustawienie długości boku o wartości 100:

```
kwadr.ustawDlBoku( 100 );
```

Od tego momentu prywatne pole `dlBoku` obiektu `kwadr` będzie miało wartość 100. Aby wyprowadzić do strumienia wyjściowego informacje o aktualnej długości boku kwadratu reprezentowanego przez `kwadr` napiszemy np.:

```
cout << "Dlugość boku kwadratu: " << kwadr.podajDlBoku();
```

Zwróćmy uwagę, że metoda `podajDlBoku` jest w istocie funkcją, której rezultatem jest wartość zapisana w polu `dlBoku`. Obiekt `kwadr` posiada również metodę wyznaczająca pole, którą przykładowo możemy wykorzystać następująco:

```
cout << "Pole kwadratu o boku: " << kwadr.podajDlBoku();
cout << " wynosi: " << kwadr.obliczPole();
```

To wystarczy aby spróbować napisać program obliczania pola kwadratu z wykorzystaniem obiektu klasy `Kwadrat`:

```
#include <cstdlib>
#include <iostream>
using namespace std;
. . .
int main() {
  double bok;
  Kwadrat kwad;
  cout << "\nObliczam pole kwadratu" << endl;
  do {
    cout << "\nDlugosc boku: " << flush;
    cin >> bok;
    if( bok < 0 )
      cout << "Dlugosc boku musi byc lb. nieujemna.\n";
  } while( bok < 0 );
  kwad.ustawDlBoku( bok );
  cout << "Pole kwadratu : " << kwadr.obliczPole() << endl;
  return EXIT_SUCCESS;
}
```

Zwróćmy uwagę, że obiekt `kwad` klasy `Kwadrat` jest odpowiedzialny za przechowywanie informacji istotnych dla kwadratu i za realizację obliczeń — my nie posługujemy się już wzorem na pole kwadratu, to kwadrat „wie” jak obliczyć własne pole. W przedstawionym wyżej programie iteracja `do-while` kontroluje poprawność wprowadzanej długości boku. Zastanówmy się chwilę nad tym fragmentem — czy to czasem obiekt `kwad` nie powinien sam weryfikować poprawności danych? Rzeczywiście, dobrze byłoby, gdyby obiekt reprezentujący kwadrat informował nas o poprawnej wartości długości boku. Można to zrealizować na wiele sposobów, tutaj zaproponujemy jeden z najprostszych.

Proponujemy zmodyfikować metodę `ustawDlBoku` tak by była funkcją, a jej rezultatem była wartość *true*, gdy wartość przekazana tej funkcji jest poprawna, jeżeli wartość ta jest niepoprawna (ujemna, równa zero) rezultatem jest wartość *false*. Niepoprawna długość jest odrzucana i przyjmuje się domyślną długość o wartości 0. Można to spróbować wykorzystać w następujący sposób:

```
#include <cstdlib>
#include <iostream>
using namespace std;
. . .
int main() {
  double bok;
  bool   poprawneDane;
  Kwadrat kwadr;
  cout << "\nObliczam pole kwadratu" << endl;
  do {
    cout << "\nDługość boku: " << flush;
    cin >> bok;
    // Próba ustawienia długości boku i jej kontrola
    poprawneDane = kwadr.ustawDlBoku( bok );
    if( !poprawneDane )
      cout << "Dlugość boku musi być lb. nieujemną.\n";
  } while( !poprawneDane );
  cout << "Pole kwadratu: " << kwadr.obliczPole() << endl;
  return EXIT_SUCCESS;
}
```

W tej wersji programu to obiekt reprezentujący kwadrat sam jest odpowiedzialny za stwierdzenie, czy przekazywana długość jest poprawna. Linia:

```
poprawneDane = kwadr.ustawDlBoku( bok );
```

zawiera przekazanie obiektowi reprezentującemu kwadrat liczby określającej długość boku, obiekt sam zadecyduje czy tę wartość przyjąć i wstawić do prywatnego pola, a rezultat funkcji `ustawDlBoku`, zapamiętany w zmiennej `poprawneDane` zawiera informacje o tym, czy dane były poprawne, czy nie. Zmienna `poprawneDane` steruje wykonaniem iteracji `do-while` — po jej zakończeniu możemy przejść do wyznaczenia pola kwadratu, co realizuje wywołanie metody `obliczPole()` obiektu `kwadr`.

Porównanie przedstawionej wcześniej nieobiektowej wersji obliczania pola z wersją obiektową, przedstawioną wyżej, skłania do natychmiastowego stwierdzenia, że wersja obiektowa jest dłuższa. Rzeczywiście tak jest. Co uzyskujemy w zamian? Wyraźne *rozdzielenie kompetencji* — program tylko wczytuje dane, przekazuje do obiektu reprezentującego kwadrat, ten weryfikuje je — zgodnie z własnymi kryteriami, oraz realizuje obliczenie pola — również zgodnie z sobie znanym wzorem. Niestety, wersja obiektowa będzie jeszcze dłuższa, bowiem klasy `Kwadrat` przecież jeszcze nie ma! Następny podrozdział poświęcony jest właśnie jej budowie.

Klasy i obiekty

Definiowanie klas polega na określaniu wzorca, według którego będą tworzone obiekty. Wzorzec ten określa dwie istotne właściwości obiektu:

- *Dane*, które obiekt będzie przechowywał.
- *Operacje*, które obiekt będzie wykonywał.

W zakresie przechowywanych *danych*, obiekt jest podobny do struktury danych zwanej *rekordem*[1], a klasa przypomina definicję typu rekordowego. Istotną różnicą jest wbudowanie w obiekt *operacji*, które może on wykonywać. Zgodnie z koncepcją programowania obiektowego, operacje te realizują *metody* wbudowane w obiekt. Jednak w języku C++ używa się nazwy *funkcja składowa* (ang. member function). Możemy stwierdzić, że *metody* w języku C++ są implementowane poprzez *funkcje składowe*.

[1] W języku C i C++ zwykle rekordy nazywa się *strukturami*.

Skoro definiowanie klasy w warstwie jej danych przypomina definiowanie typu rekordowego, pierwsza definicja klasy `Kwadrat` może mieć następującą postać:

```
class Kwadrat {
  float dlBoku;
};
```

Definicja ta informuje, że od tego momentu w programie będzie występowała klasa `Kwadrat`, posiadająca jedno *pole* będące liczbą rzeczywistą — typ `float` — o nazwie `dlBoku`. Gdybyśmy chcieli utworzyć obiekt `kwadr` klasy `Kwadrat` to piszemy oczywiście:

```
Kwadrat kwadr;
```

Możemy się spodziewać — poprzez analogię z typem rekordowym — że poprawne będzie następujące odwołanie do pola `dlBoku` obiektu `kwadr`:

```
kwadr.dlBoku = 10
```

oraz

```
cout << kwadr.dlBoku
```

Niestety odwołania te nie są prawidłowe — w przeciwieństwie do typu rekordowego, pole `dlBoku` jest *prywatne* i odwoływać się mogą do niego tylko metody klasy `Kwadrat`. W definicji klasy mogą wystąpić słowa kluczowe określające *zakres widoczności* elementów klasy. Brak jakiegokolwiek określenia zakresu widoczności elementów w klasie oznacza, że elementy te są prywatne. Możemy to zapisać jawnie, używając słowa kluczowego `private`:

```
class Kwadrat {
  private:
    float dlBoku;
};
```

Umieszczenie pola w sekcji prywatnej klasy realizuje postulat *hermetyzacji*, oznaczający tutaj ukrywanie szczegółów implementacyjnych przed otoczeniem klasy oraz ochronę danych przed niekontrolowanym dostępem. Dostęp do prywatnego pola `dlBoku` realizowany będzie przez *metody dostępowe* `podajDlBoku` oraz `ustawDlBoku`. Pamiętajmy, że metoda `ustawDlBoku` kontroluje poprawność ustawianej długości boku.

Metody `podajDlBoku` i `ustawDlBoku` muszą być dostępne dla otoczenia klasy, zatem zdefiniujemy je w sekcji *publicznej* klasy, identyfikowanej słowem `public`:

```
class Kwadrat {
  public:
    float podajDlBoku();
    bool  ustawDlBoku( float nowyBok );

  private:
    float dlBoku;
};
```

W sekcji `public` pojawiły się nazwy metod dostępowych, syntaktycznie są one *prototypami funkcji* — określają nazwy funkcji, typ ich rezultatu oraz typy parametrów[2]. Prototyp to *deklaracja funkcji*, nie zawiera ona *ciała funkcji*, czyli instrukcji, które mają być uruchomione po jej wywołaniu. Gdzie są te instrukcje? O tym za chwilę, teraz zobaczmy, że w przedstawionej powyżej klasie brak metody realizującej obliczanie pola kwadratu `obliczPole`, dopiszmy ją zatem i otrzymamy prawie kompletną definicję klasy `Kwadrat`:

[2] Przypominamy, że w języku C++ brak parametrów funkcji oznacza domyślnie wystąpienie słowa kluczowego *void*, w języku C oznacza operator '...', a więc dowolna liczbę parametrów.

```
class Kwadrat {
  public:
    float podajDlBoku();
    bool  ustawDlBoku( float nowyBok );
    float obliczPole();

  private:
    float dlBoku;
};
```

Dlaczego prawie kompletną? Brakuje przecież instrukcji stanowiących ciała trzech — zapowiedzianych prototypami — funkcji składowych. Pełna specyfikacja funkcji, składająca się z jej nagłówka i ciała jest definicją funkcji i może wyglądać następująco:

```
float Kwadrat::podajDlBoku() {
  return dlBoku;
}

bool Kwadrat::ustawDlBoku( float nowyBok ) {
  return bool( dlBoku = ( nowyBok > 0 ) ? nowyBok : 0 );
}
```

```
float Kwadrat::obliczPole() {
  return dlBoku * dlBoku;
}
```

W definicji funkcji składowych, przed każdą nazwą występuje prefiks w postaci nazwa klasy: `Kwadrat` oraz operatora '::'. Prefiks oznacza przynależność tych funkcji do klasy `Kwadrat` właśnie, a operator '::' jest operatorem zakresu. Dzięki nazwom funkcji z określeniem przynależności do klasy, możemy używać takich samych nazw w różnych klasach, a kompilator będzie wiedział, do której klasy dana funkcja należy.

Istnieje jeszcze jedna możliwość zdefiniowania funkcji składowych — jest nią umieszczenie definicji wewnątrz klasy, tam gdzie do tej pory umieszczane były prototypy funkcji:

```
class Kwadrat {
  public:
    float podajDlBoku() {
      return dlBoku;
    }
    bool ustawDlBoku( float nowyBok ) {
      return bool( dlBoku = ( nowyBok > 0 ) ? nowyBok : 0 );
    }
    float obliczPole() {
      return dlBoku * dlBoku;
    }

  private:
    float dlBoku;
};
```

Definicje funkcji wewnątrz deklaracji klasy nie wymagają stosowania prefiksów z nazwą klasy i operatorem zakresu — definiujemy funkcje w obrębie klasy i nie ma wątpliwości o ich przynależności. Obie formy — definicje poza klasą, oraz definicje wewnątrz klasy — mogą być stosowane zamiennie[3], w niniejszym opracowaniu wykorzystywana będzie forma pierwsza.

[3] W istocie istnieje pewna różnica — jest nią domyślne traktowanie funkcji zdefiniowanych w obrębie klasy jako funkcji *wstawianych*, co zostanie omówione później.

Stworzyliśmy już prototypową wersję klasy `Kwadrat`, co należy z nią zrobić? Kod klasy oraz funkcji składowych należy umieścić w miejscu trzech kropek w programie przykładowym umieszczonym na stronie 12. Teraz ten program jest kompletny i powinien działać.

Podsumowanie

W klasie definiuje się wspólne właściwości obiektów, które będą reprezentantami takiej klasy:

- typy i nazwy *pól*, w których obiekty będą przechowywać istotne dla nich dane,

- rodzaje oraz nazwy *metod*, które będą reprezentowały operacje, jakie będzie mógł wykonywać obiekt.

Właściwości — dane i metody — które opisuje klasa powinny być dostosowane do specyfiki problemu — zatem nie powinny dotyczyć wszystkich możliwych i szczególnych cech obiektów, leczy tylko tych, które w danym problemie są istotne. To pierwsza i fundamentalna zasada programowania obiektowego — *abstrakcja*.

W języku C++ zamiast pojęcia *metoda* używa się zwykle sformułowania *funkcja składowa*. Pola jak i funkcje składowe mogą mieć różny zakres widoczności, można je bowiem deklarować w dwóch[4] sekcjach:

[4] Istnieje jeszcze jeden zakres widoczności pól — *protected*, który zostanie omówiony później

- `private`, elementy umieszczone w tej sekcji mogą być wykorzystywane wyłącznie przez metody danej klasy. Elementami tymi mogą być zarówno pola jak i metody. Mówi się o nich, że są prywatne.
- `public`, elementy umieszczone w tej sekcji są dostępne również dla innych elementów programu. Mówi się o nich, że są publiczne, lub stanowią *interfejs klasy*.

Dzięki możliwości podziału elementów klasy na składowe prywatne i publiczne, możemy kierować się drugą, fundamentalną zasadą programowania obiektowego — *hermetyzacją*. Każe ona przed otoczeniem klasy ukrywać konkretny sposób przechowywania danych i szczegóły realizacji poszczególnych operacji. Zwykle zatem uprywatniamy pola klasy, dostęp do nich obudowujemy odpowiednimi metodami dostępowymi. Te ostatnie są definiowanie w sekcji publicznej klasy. Metody publiczne klasy można nieformalnie podzielić na:

- *akcesory*, metody umożliwiające pobieranie wartości uprywatnionych pól,
- *modyfikatory*, metody dokonujące modyfikacji wartości uprywatnionych pól,
- *realizatory* — metody realizujące właściwe dla danej klasy operacje.

Na koniec tego rozdziału podajemy dodatkowy komentarz dotyczący *obiektów* `cin` oraz `cin`. Są to odpowiednio obiekty klas `istream` oraz `ostream`. Oba te obiekty zdefiniowano w bibliotece obsługi strumieniowego wejścia–wyjścia, identyfikowanej przez plik nagłówkowy *iostream*. Oprócz specjalnie zdefiniowanych dla tych strumieni operatorów << oraz >>, obiekty te są wyposażone w szereg użytecznych metod, dobrze udokumentowanych w książce Plaugera [5].

[5] Plauger P.J. *Biblioteka standardowa C++*. Wydawnictwa Naukowo-Techniczne, Warszawa, Polska, 1997

Konstruktory i destruktor

Zastanówmy się, jaka będzie wartość pola kwadratu w następującym przykładzie:

```
Kwadrat kwadr;
cout << "Pole kwadratu: " << kwadr.obliczPole();
```

Definiujemy obiekt `kwadr` i bez ustalenia długości boku prosimy go o obliczenie pola. Metod `obliczPole` oczywiście wyznaczy kwadrat wartości pola `dlBoku`, tylko jaka tam jest wartość? Zgodnie z regułami języka C++, jeżeli obiekt `kwadr` będzie *automatycznym*, wartość jego pól będzie przypadkowa. Takie też będzie wyznaczone pole. Gdyby `kwadr` był obiektem *zewnętrznym*, jego pola zostałyby wyzerowane i zerowa byłaby wartość pola.

Ta niejednoznaczność nie jest dobra. Znacznie lepiej byłoby, gdyby obiekt tuż po utworzeniu inicjował wartości swoich pól, zgodnie z pewna ustaloną konwencją. W przypadku kwadratu, ta inicjalizacji mogłaby po prostu polegać na wyzerowaniu wartości pola `dlBoku`.

Ponieważ w rzeczywistości obiekty są znacznie bardziej skomplikowane niż rozważany przez nas kwadrat, inicjalizacja ich pól może być zagadnieniem złożonym. Do realizacji procesu inicjalizowania wartości pól nowo utworzonego obiektu wprowadzono do języka C++ szczególny rodzaj funkcji składowej, zwanej *konstruktorem*.

Konstruktor domyślny

Konstruktor jest specjalną funkcją, aktywowaną przez kompilator automatycznie, w momencie gdy obiekt został utworzony. Dzieje się tak zanim programista będzie mógł odwołać się do obiektu. Konstruktor ma przygotować obiekt do „życia” — inicjalizacja pól obiektu jest właśnie jedną z takich czynności.

Konstruktor to specyficzna funkcja:

- nie ma określonego typu rezultatu (nie wolno używać nawet słowa kluczowego `void` jako typu rezultatu);
- nosi taką nazwę, jak nazwa klasy;
- jest wywoływany automatycznie i zazwyczaj programista nie wywołuje jawnie konstruktora.

Uzbrojeni w tę wiedzę, spróbujmy rozszerzyć klasę `Kwadrat` o deklaracje konstruktora:

```
class Kwadrat {
  public:
    Kwadrat();
    float podajDlBoku();
    bool  ustawDlBoku( float nowyBok );
    float obliczPole();

  private:
    float dlBoku;
};
```

Deklaracja klasy wzbogaciła się o linię: `Kwadrat();` będącą deklaracją prototypową konstruktora — rzeczywiście nie został określony rezultat funkcji, jej nazwa jest taka jak nazwa klasy. Dodatkowo tak zadeklarowany konstruktor jest funkcją bezparametrową. Definicja konstruktora umieszczona zostanie poza obrębem klasy i ma następującą postać:

```
Kwadrat::Kwadrat() {
  dlBoku = 0;
}
```

Nieco dziwnie wygląda nagłówek: „`Kwadrat::Kwadrat()`”, w istocie nie ma w nim nic dziwnego, fragment „`Kwadrat::`” informuje, że mamy doczynienia z funkcją składowa klasy `Kwadrat`, natomiast reszta nagłówka „`Kwadrat()`” to po prostu nazwa funkcji składowej. Nasz konstruktor inicjalizuje długość boku kwadratu wartością 0.

Kiedy konstruktor jest wywoływany i przez kogo? Konstruktor jest wywoływany automatycznie (kompilator umieszcza w programie odpowiedni kod) tuż po utworzeniu nowego obiektu. Nieważne, czy ten obiekt jest zmienną zewnętrzną, automatyczną, czy alokowaną dynamicznie. W podanym niżej przykładzie:

```
Kwadrat a, b;
Kwadrat k[3];
```

wywołanie konstruktora nastąpi automatycznie dla obiektu `a` i `b`, oraz dla każdego z trzech elementów tablicy `k`. Każdorazowo spowoduje ono wyzerowanie pola określającego długość boku.

Konstruktor posiada jeszcze jedna cechę odróżniająca go od zwykłej funkcji — może posiadać *listę inicjalizacyjną*. Podstawowym jej zastosowaniem jest inicjowanie pól obiektu, które odbywa się przed wykonaniem ciała funkcji. Na liście może wystąpić nazwa pola, a w nawiasach okrągłych wartość inicjalizająca to pole. Alternatywna wersja konstruktora klasy `Kwadrat` może mieć następującą postać:

```
Kwadrat::Kwadrat() : dlBoku( 0 ) {}
```

Ponieważ pole `dlBoku` zostało zainicjowane na liście inicjalizacyjnej, ciało konstruktora jest puste. Która wersja konstruktora jest lepsza, ta wcześniejsza, czy ta z listą inicjalizacyjną? Programista ma tutaj sporą dowolność, jednak pewne operacje można wykonać tylko na liście inicjalizacyjnej, zatem jej stosowanie jest dobrym nawykiem i będzie tu preferowane.

Zauważmy, że wykorzystywany przez nas konstruktor nie posiada parametrów i jest wywoływany w sytuacji definiowania zmiennych bez jakichkolwiek inicjatorów. Taki rodzaj konstruktora nazywany jest *konstruktorem domyślnym* (ang. default constructor) i odpowiedzialny jest właśnie za inicjalizowanie obiektu wartościami domyślnymi.

Konstruktor ogólny

Język C++ pozwala na inicjalizowanie zmiennych na etapie ich definiowania, np. definicja `int i = 10;` inicjalizuje wartość zmiennej `i` wartością 10. Czy podobnie można postąpić ze zmienną, będącą obiektem? Rzeczywiście, na etapie definiowania obiektu, można przekazać porcję danych, które mają zainicjować pola tworzonego obiektu. Następująca definicja powołuje do życia obiekt `kwadr`, którego pole `dlBoku` ma zostać zainicjowane wartością 100:

```
Kwadrat kwadr( 100 );
```

W jaki sposób wartość 100 ma zostać wstawiona do pola `dlBoku`? Przypomnijmy, że to konstruktor ma być odpowiedzialny za inicjowanie pól obiektu.

Omówiony wcześniej konstruktor domyślny nie nadaje się do tego celu. Okazuje się jednak, że w języku C++ można zdefiniować więcej niż jeden konstruktor, również taki, który zainicjuje obiekt wartościami innymi niż domyślne. Taki konstruktor nazywany jest *konstruktorem ogólnym* (ang. general constructor). Uzupełnijmy definicję klasy kwadrat właśnie o konstruktor ogólny:

```
class Kwadrat {
  public:
    Kwadrat();
    Kwadrat( float startowyBok );
  . . .
};
```

Zauważmy, że mamy teraz dwa konstruktory, będące funkcjami o tej samej nazwie, różniące się parametrami. Taka sytuacja jest w języku C++ dozwolona, nazwy funkcji mogą być *przeciążane*. Drugi konstruktor będzie aktywowany automatycznie w opisywanej powyżej sytuacji — a więc gdy tworzymy obiekt z określeniem wartości startowej boku. Zatem definicja `Kwadrat kwadr( 100 );` spowoduje niejawne wywołanie dla obiektu `kwadr` konstruktora `Kwadrat( 100 )`. Jego zadaniem jest wstawienie wartości 100 do pola `dlBoku`, co może być zrealizowane następująco:

```
Kwadrat::Kwadrat( float startowyBok ) {
  dlBoku = startowyBok;
}
```

Wartość 100, określona na etapie definicji obiektu `kwadr`, trafia do parametru `startowyBok`, skąd jest przekazywana do pola `dlBoku` w ciele konstruktora. Można to zrealizować również z wykorzystaniem listy inicjalizacyjnej:

```
Kwadrat::Kwadrat( float startowyBok ) : dlBoku( startowyBok )
{}
```

Zauważmy jednak, że w obu tych przypadkach, konstruktory pozwolą na wstawienie do pola `dlBoku` również wartości ujemnych

— a te, jak ustaliliśmy wcześniej, nie powinny być przyjmowane. Powinniśmy skontrolować wstawianą wartość:

```
Kwadrat::Kwadrat( float startowyBok ) {
  dlBoku = ( startowyBok > 0 ) ? startowyBok : 0;
}
```

W konstruktorze można wywoływać oczywiście inne funkcje, zwróćmy uwagę, że funkcja `ustawDlBoku` posiada już zabezpieczenie przed wstawieniem wartości ujemnej, zatem nasz konstruktor parametrowy możemy ostatecznie zapisać tak:

```
Kwadrat::Kwadrat( float startowyBok ) {
  ustawDlBoku( startowyBok );
}
```

Konstruktor kopiujący

W języku C++ można inicjować wartość definiowanej zmiennej, warością innej zmiennej tego typu, np.:

```
int i = 10;
int j = i;
```

Czy podobnie można postąpić ze zmienną, będącą obiektem?

```
Kwadrat a( 100 );
Kwadrat b = a;
```

Czy można zainicjować obiekt `b`, wartością obiektu `a`? Tak, inicjalizacja polega na skopiowaniu zawartości pola `dlBoku` obiektu `a` (wartość 100 ustalona prze konstruktor ogólny), do pola `dlBoku` obiektu `b`. Gdyby pól było więcej, taką procedurę powinniśmy powtórzyć dla każdego z pól.

Wiemy już, że za inicjowanie obiektów na etapie ich tworzenia odpowiadają konstruktory. Za obsługę opisywanej sytuacji odpowiada trzeci rodzaj konstruktora — *konstruktor kopiujący* — (ang. copy constructor). Konstruktor kopiujący odpowiedzialny za skopiowanie zawartości obiektów tej samej klasy na etapie inicjalizacji. Nosi on oczywiście taka samą nazwę jak dwa pozostałe i różni się od nich parametrem — otrzymuje jeden parametr będący obiektem tej samej klasy, do której należy konstruktor. Parametr ten reprezentuje obiekt inicjalizujący, wewnątrz konstruktora można dokonać przepisania wartości z pól tego obiektu. Uzupełnijmy klasę `Kwadrat` o taki konstruktor:

```
class Kwadrat {
  public:
    Kwadrat();
    Kwadrat( float startowyBok );
    Kwadrat( Kwadrat & innyKwadrat );
  . . .
};
```

Parametr tego konstruktora to obiekt klasy `Kwadrat` o nazwie `innyKwadrat`. Znak '&' oznacza w języku C++ *referencję*, umieszczony w deklaracji parametru oznacza, że parametr ten jest przekazywany przez zmienną. Fakt ten ma istotne znaczenie — zapamiętajmy, że zapomnienie znaku oznaczającego referencję w konstruktorze kopiującym może spowodować spore problemy. Zadanie skopiowania zawartości pól może zostać zrealizowane w następujący sposób:

```
Kwadrat::Kwadrat( Kwadrat & innyKwadrat ) {
  dlBoku = innyKwadrat.dlBoku;
}
```

Alternatywnie, można wykorzystać listę inicjalizacyjną konstruktora kopiującego:

```
Kwadrat::Kwadrat( Kwadrat & innyKwadrat )
  : dlBoku( innyKwadrat.dlBoku ) {}
```

Przedstawiony powyżej przykład konstruktora kopiującego pozwala zrozumieć, jaka jest jego koncepcja i jak on działa. Jednak w rozważanym przypadku nie trzeba definiować konstruktora kopiującego. Jest tak dlatego, że w przypadku jego braku, kompilator zrealizuje inicjalizację kopiując pole po polu, zawartość obiektu inicjującego do obiektu inicjalizowanego – a w naszym przypadku w zupełności to wystarczy. Zdefiniowanie konstruktora kopiującego jest konieczne w przypadku sytuacji, gdy takie kopiowanie nie wystarcza, a tak jest np. w przypadku pól wskaźnikowych, odwołujących się do dynamicznie alkowanych obszarów pamięci operacyjnej.

Konstruktor rzutujący

W języku C++ można inicjować wartość definiowanej zmiennej, wartością innej zmiennej, innego typu, jeżeli istnieje droga konwersji wartości, np.:

```
int i = 10;
float j = i;
```

Załóżmy, że istnieje klasa `Prostokąt`, analogiczna do klasy `Kwadrat`. Czy można zainicjować obiekt klasy `Prostokąt` obiektem klasy `Kwadrat`? Kwadrat jest przecież prostokątem, zatem inicjowany prostokąt będzie po prostu prostokątem o równych bokach.

```
Kwadrat a( 100 );
Prostokąt b = a;
```

Domyślamy się już, że można to zrobić, i że będzie za to odpowiedzialny odpowiedni konstruktor. Taki konstruktor nazywa się *konstruktorem rzutujacym* (ang. cast constructor). Ale uwaga — ten konstruktor będzie elementem klasy `Prostokąt`, bo to konwersję na obiekt tej klasy przewidujemy, dla obiektu klasy `Kwadrat`. Konwersja odwrotna nie jest rozsądna.

Zatem by przedstawić koncepcję konstruktora rzutującego, musimy najpierw stworzyć klasę `Prostokąt`.

Klasa Prostokąt

Dla klasy prostokąt najistotniejsze będą długości dwóch boków — stanowią one szerokość i wysokość prostokąta. Będą to podstawowe informacje istotne dla prostokąta, prezentuje to rys. 5.

Model projektowy prezentuje rys. 6, zawiera on podstawowe elementy — pola, akcesory, modyfikatory, realizator. Diagram ten nie zawiera jeszcze konstruktorów.

Pełny kod źródłowy klasy `Prostokat` zawarty jest w dalszej części tego podrozdziału.

Model analityczny

Rysunek 5: Model analityczny

Rysunek 6: Model projektowy

Model projektowy

Prostokat
- szer: double - wys : double
+ podajWysokosc(): double + ustawWysokosc(nowaWys : double) : bool + podajSzerokosc(): double + ustawSzerokosc(nowaSzer : double) : bool + obliczPole() : double

```
class Prostokat {
  public:
    Prostokat();
    Prostokat( double startWys, double startSzer );
    Prostokat( Kwadrat & jakisKwadrat );
    double podajWysokosc();
    bool   ustawWysokosc( double nowaWys );
    double podajSzerokosc();
    bool   ustawSzerokosc( double nowaSzer );
    double obliczPole();

  private:
    double szer;
    double wys;
};

Prostokat::Prostokat() : wys( 0 ), szer( 0 ) {}

Prostokat::Prostokat( double startWys, double startSzer ) {
  ustawWysokosc( startWys );
  ustawSzerokosc( startSzer );
```

```
}

Prostokat::Prostokat( Kwadrat & jakisKwadrat )
  : szer( jakisKwadrat.podajDlBoku() ),
    wys( jakisKwadrat.podajDlBoku() ) {}

double Prostokat::podajWysokosc() {
  return wys;
}

bool Prostokat::ustawWysokosc( double nowaWys ) {
  return bool( wys = ( nowaWys > 0 ) ? nowaWys : 0 );
}

double Prostokat::podajSzerokosc() {
  return szer;
}

bool Prostokat::ustawSzerokosc( double nowaSzer ) {
  return bool( szer = ( nowaSzer > 0 ) ? nowaSzer : 0 );
}

double Prostokat::obliczPole() {
  return wys * szer;
}
```

Konstruktor rzutujący raz jeszcze

Klasa `Prostokat` posiada zdefiniowany konstruktor rzutujący pozwalający na zainicjowanie danymi kwadratu, obiektu reprezentującego prostokąt. Konstruktor rzutujący klasy `Prostokat` otrzymuje jeden parametr referencyjny innego typu — w naszym przypadku nazywa się on `jakisKwadrat`. Przetransponowanie danych kwadratu do prostokąta polega na zainicjowaniu jego wysokości i szerokości długością boku kwadratu, można to zrobić następująco:

```
Prostokat::Prostokat( Kwadrat & jakisKwadrat )
  : szer( jakisKwadrat.podajDlBoku() ),
    wys( jakisKwadrat.podajDlBoku() ) {}
```

W przypadku braku zaufania do danych zawartych w obiekcie klasy `Kwadrat`, można użyć w konstruktorze rzutującym funkcji ustawiających szerokość i wysokość, te realizują odpowiednie czynności kontrolne:

```
Prostokat::Prostokat( Kwadrat & jakisKwadrat ) {
  ustawSzerokosc( jakisKwadrat.podajDlBoku() );
  ustawWysokosc( jakisKwadrat.podajDlBoku() );
}
```

Destruktor

Konstruktory zapewniają automatyczne wykonywanie pewnych czynności dla obiektu, który właśnie powstał. Język C++ oferuje mechanizm komplementarny — automatyczne wykonywanie czynności dla obiektu, który zostanie za chwilę usunięty. Dla tej sytuacji przewidziano również specjalną funkcję, nazywaną *destruktorem*. W przeciwieństwie do konstruktorów w danej klasie przewiduje się istnienie tylko jednego destruktora.

Typowa rola destruktora to wykonanie czynności porządkowych, przed usunięciem obiektu z pamięci operacyjnej. W przypadku klas `Kwadrat` czy `Prostokat` trudno wskazać sensowne wykorzystanie destruktora. Przeanalizujmy jego działanie na innym przykładzie.

Załóżmy, że potrzebujemy obiektu, który będzie rejestrował czas rozpoczęcia programu — rozpoczęcia wykonania funkcji `main`, a tuż przed jej zakończeniem wyliczy ile od tego czasu upłynęło sekund i wyprowadzi tę informację do strumienia wyjściowego programu. Niech klasa tego obiektu nazywa się `RejestratorCzasu`, a jego wykorzystanie będzie następujące:

```
int main() {
  RejestratorCzasu r;
  int i;
  cout << "Wpisz liczbe: ";
  cin >> i;
  return EXIT_SUCCESS;
}
```

Program prosi użytkownika o wpisanie liczby, przed zakończeniem programu ma on ma wyświetlić, ile trwała ta operacja. Osiągniemy to bez jakiejkolwiek modyfikacji przedstawionej funkcji `main`. Jak to możliwe?

Pierwsza linia tej funkcji zawiera definicję obiektu `r` klasy `RejestratorCzasu`. Zatem tuż po rozpoczęciu funkcji `main` zmienna ta jest tworzona. Przypomnijmy, że tuż po utworzeniu obiektu `r`, wywoływany jest *automatycznie jego konstruktor*, w naszym przypadku to konstruktor domyślny. Zadaniem konstruktora będzie zarejestrowanie aktualnego stanu zegara systemowego. Obiekt `r` jest automatyczny, zostanie zatem usunięty z pamięci operacyjnej tuz przed zakończeniem funkcji `main`. Wtedy też, zostanie dla tego obiektu *automatycznie wywołany destruktor*. Jego zadaniem będzie zarejestrowanie stanu zegara systemowego w tym momencie, oraz wyznaczenie różnicy pomiędzy tym stanem zegara, a stanem zapamiętanym w konstruktorze. Różnica da przybliżony czas wykonania funkcji `main`.

Klasa `RejestratorCzasu` będzie posiadał tylko jedno pole prywatne, przechowujące stan zegara systemowego zarejestrowany w konstruktorze. Niech pole nazywa się `start` i jest typu `clock_t`, zdefiniowanego w pliku nagłówkowym *ctime*. Klasa będzie zawierała tylko dwie funkcje składowe — konstruktor domyślny i destruktor.

```
class RejestratorCzasu {
  public:
    RejestratorCzasu() {
      start = clock();
    }
    ~RejestratorCzasu() {
       clock_t end = clock();
       cout << "Czas: ";
       cout << int( ( end - start )/CLK_TCK );
       cout << " sek.";
    }

  private:
    clock_t start;
};
```

Konstruktor domyślny, o oczywistej nazwie `RejestratorCzasu`, wywołuje funkcję biblioteczną `clock()`, której rezultatem jest aktualny stan zegara systemowego. I to wszystko. Resztę robi destruktor, zauważmy, że jego nazwa to `~RejestratorCzasu`, nosi więc nazwę klasy poprzedzoną znakiem tyldy, nie ma określonego rezultatu i nie otrzymuje parametrów. Destruktor zapamiętuje w lokalnej zmiennej `end` stan zegara w momencie rozpoczęcia działania destruktora. Różnica `end - start` określa liczbę impulsów zegarowych, po podzieleniu ich przez stałą `CLK_TCK` otrzymujemy przybliżoną liczbę sekund, jaką trwało wykonanie bloku.

Parametry domyślne

Parametr domyślny to wartość określona na etapie deklaracji funkcji, która zostanie automatycznie wstawiona do *parametru formalnego*, jeżeli dana funkcja zostanie wywołana bez odpowiedniego *parametru aktualnego*. Parametry domyślne dotyczą funkcji składowych klas jak i funkcji niezwiązanych z klasami. Prześledźmy to na przykładzie, załóżmy, że w prototypie funkcji określono dwa ostatnie parametry, jako domyślne:

```
void fun( int i, float f = 0, char c = 'A' );
```

Taką funkcję można wywoływać z pominięciem, końcowych parametrów, ich wartość zostanie ustalona automatycznie zgodnie z wartościami domyślnymi:

```
fun( 10 );             // i == 10, f == 0,    c == 'A'
fun( 20, 3.15 );       // i == 20, f == 3.15, c == 'A'
fun( 30, 22.1, 'Z' );  // i == 30, f == 22.1, c == 'Z'
```

Jeżeli stosujemy prototypy funkcji, wartości parametrów domyślnych określa się właśnie w prototypie, w definicji funkcji już nie występują.

```
void fun( int i, float f = 0, char c = 'A' );
```

```
. . .
void fun( int i, float f, char c ) {}
```

Parametry domyślne można wykorzystać w celu zmniejszenia liczby konstruktorów. Zauważmy, że rozważana wcześniej klasa `Kwadrat` posiadała konstruktor *domyślny* i *ogólny*:

```
class Kwadrat {
  public:
    Kwadrat();
    Kwadrat( float startowyBok );
  . . .
};
. . .
Kwadrat::Kwadrat() : dlBoku( 0 ) {}

Kwadrat::Kwadrat( float startowyBok ) {
  dlBoku = ( startowyBok > 0 ) ? startowyBok : 0;
}
```

Zauważmy, że po wprowadzeniu parametru domyślnego do konstruktora ogólnego, bedzie on mógł wypełniać również rolę konstruktora domyślnego, który stanie się zbędny:

```
class Kwadrat {
  public:
    Kwadrat( float startowyBok = 0 );
  . . .
};
. . .
Kwadrat a;       // Wywołanie Kwadrat() z param. domyśl. 0
Kwadrat b( 10 ); // Wywołanie Kwadrat( 10 )
```

Podsumowanie

Czas na krótkie podsumowanie informacji o konstruktorach — niżej zakładamy, że rozważania dotyczą pewnej klasy o nazwie `A`.

Konstruktor domyślny (ang. default constructor):

- Jest bezparametrowy `A()`, lub posiada wszystkie parametry będące parametrami domyślnymi: `A( arg1 = wart1, arg2 = wart2, ... )`.
- Jednoczesne wystąpienie obu powyższych form spowoduje błąd kompilacji.
- Inicjuje obiekty, deklarowane lub (bądź tworzone) bez parametrów.
- Dotyczy to również obiektów będących elementami tablicy.

```
A  a, b, c;    // Aktywacja: a.A(), b.A(), c.A()
A  tab[ 10 ];  // Aktywacja: A() dla każdego z 10-ciu elementów
               // tab: tab[ 0 ].A(), tab[ 1 ].A(), itd... .
```

Konstruktor ogólny (ang. general constructor):

- Jest to podstawowy konstruktor przeznaczony do inicjowania obiektów na etapie ich deklaracji czy też tworzenia: `A( arg1, arg2, ... )`.
- Argumenty określają zwykle wartości jakie mają być przypisane określonym polom obiektu.
- Konstruktorów głównych może być więcej, mogą one zawierać również parametry domyślne.
- Szczególnym przypadkiem jest konstruktor posiadający tylko parametry domyślne, staje się on wtedy konstruktorem domyślnym.

```
A( int a, float b, char * c  = NULL );
A obj1( 2, 3.4, "Ala");
A obj2( 1, 0.0 );
A obj3( 5, 5.5, "Pięć");
```

Konstruktor kopiujący (ang. copy constructor):

- Jest potrzebny jedynie wtedy, gdy przewidziana jest inicjalizacja obiektu danej klasy innym obiektem tejże klasy, przyjmuje różne postacie:

```
A( A & obj )
A( A & obj, arg1 = wart1, ... )
A( const A & obj )
A( const A & obj, arg1 = wart1, ... )
```

- W przypadku braku konstruktora kopiującego, kompilator zrealizuje inicjowanie nowotworzonego obiektu, kopiując pole po polu, wartości z obiektu inicjalizującego.
- Stosowanie konstruktora kopiującego jest dobrą, programistyczną praktyką. Dzięki jawnie zdefiniowanym konstruktorom programista ma kontrolę nad kopiowaniem wartości, występujących w wielu, czasem zaskakujących sytuacjach.

```
A obj1;
A obj2 = obj1;  // Uaktywnienie obj2.A( obj1 )
A obj3( obj2 ); // Uaktywnienie obj3.A( obj2 )
```

Konstruktor rzutujący (ang. cast constructor):

- Posiada jeden parametr będący referencją obiektu innej klasy. Innych argumentów może nie być lub powinny być one argumentami domyślnymi: `A( B & obj )` lub `A( const B & obj )`.
- Jest stosowany wszędzie tam, gdzie należy zainicjować obiekt pewnej klasy wartością obiektu innej klasy.
- Programista może dzięki konstruktorowi rzutującemu określić w jaki sposób informacje zapisane w obiekcie klasy B mają zostać odwzorowane (przepisane) w obiekcie klasy A.

```
A obj1;
B obj2( obj1 ); // Uaktywnienie obj2.B( obj1 )
```

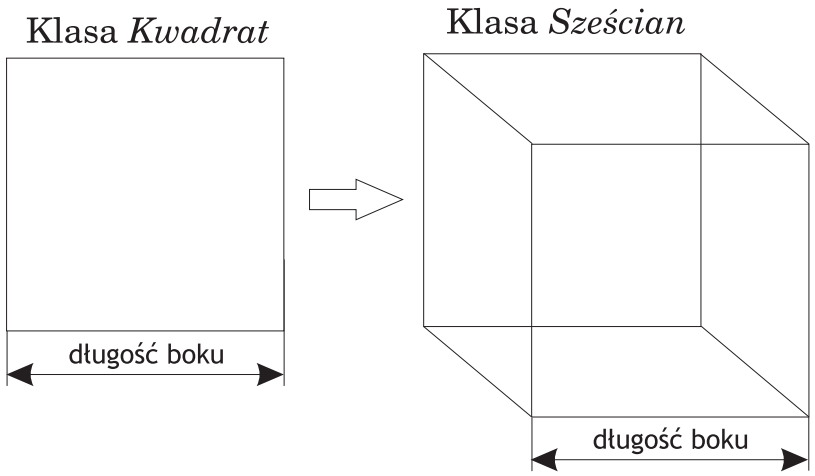
Dziedziczenie w programowaniu obiektowym

Koncepcja *dziedziczenia* (ang. inheritance) pozwala na budowanie nowych klas na podstawie klas już istniejących. Te nowe klasy, nazywane są *klasami pochodnymi*, zaś klasy stanowiące podstawę dziedziczenia, nazywamy *klasami bazowymi*. Każda klasa pochodna dziedziczy wszystkie właściwości klasy bazowej, rozszerzając ją o nowe pola i/lub metody.

Dziedziczenie jest zatem procesem tworzenia klas potomnych (ang. derivation), pozwala ono urzeczywistnić pomysł *powtórnego wykorzystania kodu*. Koncepcja ta w oryginale nosi angielską nazwę *code reusability*. Zakłada ona, że pewnych klas nie trzeba tworzyć od nowa, o ile istnieją takie, które można rozszerzyć lub zaadaptować do stojących przed programistą nowych zadań.

Od kwadratu do sześcianu — przykład dziedziczenia

Załóżmy, że naszym zadaniem jest napisanie programu obliczającego pole i objętość *sześcianu*. Sześcian jest oparty na kwadracie, o jego polu i objętości decyduje długość boku jednej ze ścian. A każda z nich jest kwadratem. Można zatem założyć, że sześcian to specyficzny kwadrat — wyciągnięty w przestrzeni, obdarzony trzecim wymiarem. Ilustruje to rys. 7.



Rysunek 7: Od Kwadratu do Sześcianu — koncepcja dziedziczenia

W poprzednim rozdziale zdefiniowaliśmy klasę `Kwadrat`, definiowała ona kwadrat, jako figurę geometryczną określoną długością boku, pamiętaną w polu o nazwie `dlBoku`. Klasa ta definiowała również funkcję składową `obliczPole`, obliczającą pole kwadratu. Nasuwa się pytanie — czy można wykorzystać istniejący już kod klasy `Kwadrat` do utworzenia klasy reprezentującej sześcian? Niech ta klasa nazywa się właśnie `Szescian` i niech zawiera funkcje wyznaczania łącznego pola powierzchni wszystkich ścian: `obliczPole`, oraz funkcję wyznaczania objętości: `obliczObjetosc`.

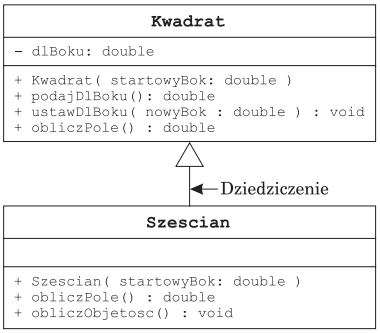
Rzeczywiście, klasa `Kwadrat` może posłużyć jako klasa bazowa do opracowania klasy `Szescian`. Koncepcyjnie wydaje się to poprawne

— *bryła* sześcian powstaje na bazie *figury* będącej kwadratem a do opisu parametrów sześcianu wystarczy długość boku kwadratu. Niestety, funkcja `obliczPole` klasy `Kwadrat` oblicza pole kwadratu a nie sześcianu — trzeba będzie coś z tym zrobić. Klasa `Kwadrat` nie posiada również funkcji obliczającej objętość, trzeba ją będzie zdefiniować. Rozpocznijmy jednak od przypomnienia ostatecznej wersji klasy `Kwadrat`:

```
class Kwadrat {
  public:
    Kwadrat( double startowyBok = 0 );
    double podajDlBoku();
    bool  ustawDlBoku( double nowyBok );
    double obliczPole();

  private:
    double dlBoku;
};
```

Funkcje składowe tej klasy mogą mieć następującą postać:

```
Kwadrat::Kwadrat( double startowyBok ) {
  ustawDlBoku( startowyBok );
}

double Kwadrat::podajDlBoku() {
  return dlBoku;
}

bool Kwadrat::ustawDlBoku( double nowyBok ) {
  return bool( dlBoku = ( nowyBok > 0 ) ? nowyBok : 0 );
}

double Kwadrat::obliczPole() {
  return dlBoku * dlBoku;
}
```



Rysunek 8: Od Kwadratu do Sześcianu — diagram hierarchii klas

Koncepcję dziedziczenia ilustruje rys. 8. Każdy obiekt klasy `Szescian` będzie zawierał wszystkie elementy klasy `Kwadrat` oraz dodatkowo funkcję obliczania objętości. Przypadek funkcji `obliczPole` wymaga dodatkowego komentarza.

Funkcja obliczania pola zdefiniowania w klasie opisu kwadratu wyznacza pole kwadratu we właściwy dla tej figury sposób. Niestety, po zastosowaniu dziedziczenia, funkcja ta w klasie reprezentującej sześcian działałaby wadliwie — wyznaczałaby dla sześcianu pole wg. wzoru dla kwadratu. Dlatego dokonujemy w klasie `Szescian` *redefinicji* funkcji obliczania pola — definiujemy `obliczPole` ponownie, tak by działała według wzoru właściwego dla sześcianu.

Zobaczmy, jak buduje się klasę pochodną w języku C++. Rozpocznijmy od zdefiniowania pustej klasy `Szescian`:

```
class Szescian : public Kwadrat {};
```

W definicji klasy pojawił się nowy zapis — fraza „`: public Kwadrat`" oznacza, że klasa `Szescian` powstaje, dziedzicząc wszystkie pola i metody klasy `Kwadrat`. Słowo kluczowe `public` oznacza, że te składowe klasy, które były publiczne w klasie `Kwadrat`, są również publiczne w klasie `Szescian`, a składowe prywatne pozostają prywatnymi[6].

[6] Inaczej mówiąc — w klasie pochodnej zostaje zachowana taka widoczność składowych klasy, jak obowiązywała w klasie bazowej. Tak być nie musi, lecz na tym etapie pomińmy inne możliwości, aby nie wprowadzać niepotrzebnego zamieszania.

Rozbudowa klasy opisu sześcianu

Spróbujmy rozszerzyć definicję klasy `Szescian` o nowe elementy:

```
class Szescian : public Kwadrat {
  public:
    double obliczPole();
    double obliczObjetosc();
};
```

Zadeklarowaliśmy, że klasa `Szescian` będzie posiadała funkcję `obliczObjetosc` oraz własną wersję funkcji obliczania pola `obliczPole`. Spróbujmy zdefiniować pierwszą wersję tych funkcji:

```
double Szescian::obliczPole() {
  return 6 * podajDlBoku() * podajDlBoku();
}

double Szescian::obliczObjetosc() {
  return podajDlBoku() * podajDlBoku() * podajDlBoku();
}
```

Zauważmy, że `dlBoku` jest polem prywatnym, zatem funkcje klasy pochodnej nie mają do niego dostępu. Dlatego w obliczeniach pola i objętości wywoływana jest funkcja `podajDlBoku`. Pole sześcianu to rzeczywiście sześciokrotność pola jednego boku, będącego kwadratem. A objętość to iloczyn pola podstawy i długości boku. Zwróćmy uwagę na użyte sformułowanie *sześciokrotność pola jednego boku* i *iloczyn pola podstawy i długości boku*. Czy można by w naszych obliczeniach użyć funkcji obliczania pola, zdefiniowanej w klasie `Kwadrat`? Przecież to ona właśnie służy do wyznaczania pola jednej ściany sześcianu, ta bowiem jest kwadratem....

Okazuje się, że w funkcjach klasy pochodnej możemy używać funkcji klasy bazowej, również tych *przedefiniowanych*. W przypadku tych funkcji pojawia się problem — ich nazwy są jednakowe. Spowodować to może problem przy rozstrzyganiu, która z wersji nas interesuje. Przypomnijmy jednak, że możemy korzystać z operatora zakresu i nazwy klasy — i tak użyjemy prefiksu `Kwadrat::` aby stwierdzić, że interesuje nas wersja funkcji z klasy `Kwadrat`, natomiast prefiksu `Szescian::` dla funkcji z tej klasy właśnie. Wykorzystamy tę właściwość do napisania drugiej — zdaniem autora lepszej — wersji funkcji obliczania pola i objętości sześcianu:

```
double Szescian::obliczPole() {
  return 6 * Kwadrat::obliczPole();
}

double Szescian::obliczObjetosc() {
  return Kwadrat::obliczPole() * podajDlBoku();
}
```

Dlaczego te wersje są lepsze? Po pierwsze — oddają one wspomniane wcześniej, intuicyjnie zrozumiałe sposoby wyznaczania pola i objętości sześcianu, wykorzystujące fakt iż bok (podstawa) jest kwadratem. Po drugie — korzystamy ze wzorów zdefiniowanych w klasie `Kwadrat`, nie przepisując ich ponownie. Ostatecznie — klasa `Szescian` zajmuje się realizacją własnych obliczeń, wykorzystując odziedziczone właściwości klasy `Kwadrat`.

Dziedziczenie a konstruktory

Klasa pochodna dziedziczy wszystkie składowe każdej klasy podstawowej, z wyjątkiem konstruktorów i destruktorów[7]. Przykładowo, nie powiedzie się próba skompilowania przedstawionego niżej kodu:

[7] Klasa pochodna nie dziedziczy również przeciążonych operatorów przypisania — tych jednak jeszcze w tym opracowaniu nie omawialiśmy.

```
Szescian kostka( 10 );
cout << "Szescian o boku: " << kostka.podajDlBoku() << endl;
cout << "       Objetosc: " << kostka.obliczObjetosc() << endl;
cout << "   Powierzchnia: " << kostka.obliczPole() << endl;
```

W klasie `Kwadrat` zdefiniowano konstruktor ogólny, nie zostanie on jednak aktywowany automatycznie dla obiektu klasy `Szescian`. W klasie pochodnej programista powinien zdefiniować konstruktory na nowo. Istnieje pewne rozluźnienie tej zasady, dotyczące konstruktorów domyślnych (bezparametrowych). Rozluźnienie to jest jednak mocno dyskusyjne, sam twórca języka — Brajne Stroustrup — namawia do definiowania również konstruktorów domyślnych klas pochodnych. Zapamiętajmy zatem: przy tworzeniu klasy pochodnej, programista zdefiniować powinien wszystkie niezbędne dla niej konstruktory.

Przy budowaniu klas pochodnych kierujemy się następującą zasadą: w klasie pochodnej definiujemy metody do obsługi nowych pól, obsługę pól odziedziczonych realizujemy z wykorzystaniem metod odziedziczonych. Mimo, że konstruktory klasy pochodnej nie są jawnie dziedziczone, programista ma do nich dostęp. Może zatem aktywować je i przy ich użyciu inicjować odziedziczone pola klasy bazowej. Deklaracja klasy `Szescian` z konstruktorem ogólnym (realizującym również funkcje konstruktora domyślnego) ma następującą postać:

```
class Szescian : public Kwadrat {
  public:
    Szescian( double startowyBok = 0 );
```

```
    double obliczPole();
    double obliczObjetosc();
};

Szescian::Szescian( double startowyBok ) : Kwadrat( startowyBok )
{}
```

Jakie zadanie realizuje ten konstruktor? Otrzymuje w postaci parametru startową długość boku, która powinna być wstawiona do pola `dlBoku`, zdefiniowanego w klasie `Kwadrat`. Przypomnijmy, że zadanie to realizuje konstruktor tej klasy, dbając o poprawność wstawianej wartości. W języku C++ zwykle nie wywołuje się jawnie konstruktorów, stąd mówimy raczej o *aktywowaniu* konstruktora na liście inicjalizacyjnej, a nie jego wywołaniu. Lista inicjalizacyjna jest legalnym miejscem aktywowania konstruktora klasy bazowej. Wykorzystanie konstruktora klasy bazowej polega zatem na umieszczeniu jego nazwy na liście inicjalizacyjnej konstruktora klasy pochodnej.

Po aktywowania konstruktora klasy `Szescian`, pierwszą czynnością będzie przetworzenie jego listy inicjalizacyjnej, a to spowoduje aktywację konstruktora klasy `Kwadrat`, któremu przekazujemy wartość parametru `startowyBok`. Dopiero po przetworzeniu listy inicjalizacyjnej, wykonane zostanie ciało konstruktora. W naszym przypadku jest ono puste, bowiem nie przewidujemy żadnych dodatkowych czynności w konstruktorze klasy `Szescian`[8]. W ten sposób, najpierw aktywowany zostanie konstruktor klasy bazowej, a potem dopiero ciało konstruktora klasy pochodnej.

I tak w podanym wcześniej przykładzie dla zdefiniowanego obiektu `kostka` aktywowany zostanie konstruktor parametrowy `Szescian` z parametrem 10. Konstruktor ten uaktywni — umieszczony na liście inicjalizacyjnej — konstruktor klasy `Kwadrat`, który ustawi startową wartość długości boku. Następnie zostaną wywołane funkcje obliczania pola i objętości — oczywiście wywołana zostanie funkcja `obliczPole` zdefiniowana w klasie `Szescian`.

[8] Zwykle jednak konstruktor klasy pochodnej, po aktywowaniu konstruktora klasy bazowej, zajmuje się inicjalizacją dodatkowych pól klasy pochodnej. W przypadku klasy *Sześcian* takie pola nie występują.

Redefinicja metod

Redefiniowanie funkcji składowych w klasach pochodnych pozwala na modyfikowanie ich działania tak, by było ono zgodne z wymaganiami stawianymi nowej klasie. W tak przedefiniowanej funkcji, można używać funkcji oryginalnej — odziedziczonej z klasy bazowej. Jednak takie działanie niesie ze sobą pewne ryzyko. Załóżmy, że programista piszący funkcję `obliczPole` pomylił się i napisał ją następująco:

```
double Szescian::obliczPole() {
  return 6 * obliczPole();
}
```

Cóż takiego napisał? Ano napisał, że powierzchnia sześcianu to sześciokrotność powierzchni… sześcianu! Brak nazwy kwalifikowanej `Kwadrat::` — a właśnie na jej pominięciu polegał błąd

programisty — spowoduje, że funkcja `obliczPole` będzie wywoływała samą siebie! Mamy tutaj swoistą, niezamierzoną rekurencję, bez warunku jej zakończenia. Jak się zachowa program w takiej wersji? Proponujemy czytelnikom, aby sprawdzili to samodzielnie.

Zakres widoczności protected

W klasie pochodnej nie ma bezpośredniego dostępu do pól prywatnych klasy bazowej. Przypomnijmy, że aby np. obliczyć objętość sześcianu, musieliśmy odwoływać się do pola `dlBoku` klasy `Kwadrat` za pośrednictwem akcesora — funkcji `podajDlBoku`:

```
double Szescian::obliczObjetosc() {
  return Kwadrat::obliczPole() * podajDlBoku();
}
```

W niektórych sytuacjach jest to niewygodne i nienaturalne. W rozważanym przypadku tak jest rzeczywiście — funkcje składowe klasy `Szescian` nie mogą się odwoływać bezpośrednio do podstawowej informacji opisującej tę figurę! Można jednak tę sytuację zmienić i doprowadzić do tego, że klasa pochodna `Szescian` będzie miała dostęp do pola `dlBoku`. Wymaga to zmiany w deklaracji klasy `Kwadrat` — wystarczy zamienić nazwę sekcji, w której zadeklarowane jest pole `dlBoku` z `private` na `protected`:

- Składowe zadeklarowane jako `protected` są dostępne dla obiektów wszystkich klas pochodnych (tak jak składowe `public`).
- Składowe zadeklarowane jako `protected` są niedostępne dla obiektów innych, niezaprzyjaźnionych klas (tak jak składowe `private`).
- Specyfikator `protected` działa jak `private`, z tym wyjątkiem, że obiekty klas pochodnych otrzymują dostęp do składowych `protected` klasy bazowej.
- Pola i funkcje zadeklarowane w sekcji `protected` nazywane są *chronionymi*.

Deklaracja pola `dlBoku` jako chronionego wymaga zatem wykorzystania słowa kluczowego `protected`:

```
class Kwadrat {
  . . .
  protected:
    double dlBoku;
};
```

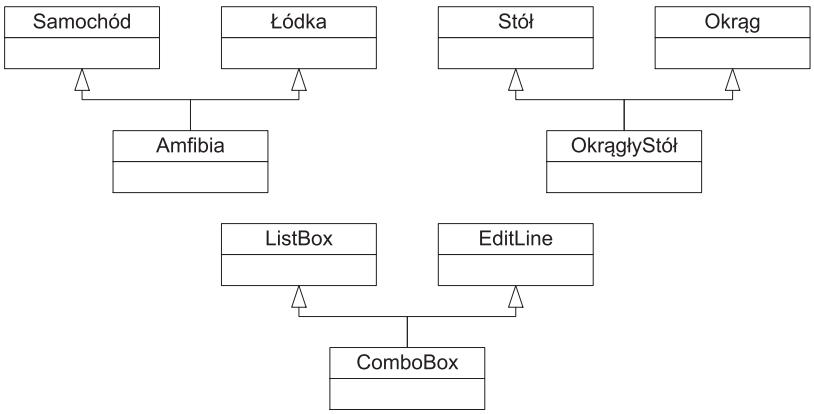
Do tak zadeklarowanego pola można się bezpośrednio odwoływać w funkcjach klasy pochodnej:

```
double Szescian::obliczObjetosc() {
  return Kwadrat::obliczPole() * dlBoku;
}
```

W rozważanym przypadku takie bezpośrednie odwołanie jest wygodniejsze i naturalniejsze. Stosowanie pól chronionych wymaga przewidywania — w klasie bazowej należy użyć słowa kluczowego `protected` zamiast `private`.

Dziedziczenie jedno i wielobazowe

Przedstawiony w poprzednim podrozdziale przykład prezentował *dziedziczenie jednobazowe* — klasa pochodna posiada *jedną klasę bazową*. Często jednak zdarza się, że klasa, która chcemy utworzyć w naturalny sposób dziedziczy właściwości z więcej niż jednej klasy — przykłady prezentuje rys. 9. Takie dziedziczenie nazywamy *wielobazowym*, pozwala ono na tworzenie klas potomnych, dziedziczących z więcej niż jednej klasy bazowej. Mając np. dwie, zaimplementowane klasy, można zbudować nową klasę, dziedziczącą jednocześnie właściwości obu tych klas.



Rysunek 9: Przykłady dziedziczenia wielobazowego

Dziedziczenie wielobazowe przedstawimy na przykładzie klasy reprezentującej okrągły stół, dziedziczącej właściwości po klasach reprezentującej okrąg i stół. Załóżmy, że dane są dwie klasy[9]:

[9] Dla poprawy czytelności przedstawiono tylko niezbędne elementy klas, brakuje w nich np. akcesorów i modyfikatorów.

```
class Okrag {
  public:
    Okrag( double r = 0 ) : promien( r )
    {}
    double obliczPole() {
       return M_PI * promien * promien;
    }

  protected:
    double promien;
};

class Stol {
  public:
    Stol( int ln = 0 ) : liczbaNog( ln )
    {}
```

```
    int podajLiczbeNog() {
      return liczbaNog;
    }

  protected:
    int liczbaNog;
};
```

Wydaje się, że znaczenie pól i funkcji składowych obu klas jest intuicyjnie zrozumiałe. Na bazie tych dwóch klas tworzymy klasę `OkraglyStol`:

```
class OkraglyStol : public Okrag, public Stol {
  public:
    OkraglyStol( double r = 0, int ln = 0 )
      : Okrag( r ), Stol( ln ) {}
};
```

Klasa ta dziedziczy wszystkie pola i funkcje składowe obu klas bazowych. Możemy zatem dla obiektu klasy `OkraglyStol` zarówno wykorzystywać funkcję `obliczPole` odziedziczoną z klasy `Okrag` jak i funkcję `podajLiczbeNog` odziedziczoną z klasy `Stol`:

```
OkraglyStol stolik( 1, 3 ); // 1 - promien, 3 - lb. nog
cout << "\nLiczba nog : " << stolik.podajLiczbeNog();
cout << "\nPowierzchnia stolu : " << stolik.obliczPole();
```

Zwróćmy uwagę na definicję obiektu `stolik` — aktywowany jest dla niego konstruktor klasy `OkraglyStol`. Ponieważ klasa ta dziedziczy po dwóch klasach, na liście inicjalizacyjnej tego konstruktora umieszczamy odwołania do dwóch konstruktorów. W trakcie przetwarzania listy inicjalizacyjnej następuje aktywowanie obu tych konstruktorów.

Kolejność aktywowania konstruktorów dla obiektu klasy pochodnej wynika z kolejności występowania nazw klas bazowych w deklaracji tej klasy. Nie jest istotna kolejność ich umieszczenia na liście inicjalizacyjnej konstruktora klasy pochodnej.

Przedstawiony przykład prezentuje elastyczność dziedziczenia wielobazowego — klasa pochodna nabyła w naturalny sposób cechy klas bazowych, osiągnęliśmy to bez pisania żadnego kodu (z wyjątkiem konstruktora oczywiście). Niestety z dziedziczeniem wielobazowym wiąże się wiele problemów — np. w przypadku gdy klasy bazowe mają pola o jednakowych nazwach, gdy klasy bazowe same dziedziczą właściwości po wspólnej klasie bazowej. Wszystko to mocno komplikuje praktykę wykorzystania dziedziczenia wielobazowego, a omówienie tych problemów przekracza ramy tego opracowania.

Konstruktory i destruktory a hierarchia klas

Dziedziczenie jest jednym z podstawowych i najczęściej stosowanych mechanizmów programowania obiektowego. Jego wykorzystanie

prowadzi często do powstania rozbudowanej hierarchii klas — na jej szczycie jest klasa, która nie dziedziczy właściwości z żadnej innej klasy.

Budując nowe klasy z wykorzystaniem dziedziczenia, programista powinien zadbać o zdefiniowanie konstruktorów dla każdej nowej klasy. Jak wspomniano wcześniej, zwyczajowo konstruktor klasy pochodnej aktywuje konstruktor swej klasy bazowej poprzez umieszczenie jego nazwy na liście inicjalizacyjnej. Konstruktor klasy bazowej uruchamia się zatem *przed* ciałem konstruktora klasy pochodnej. Można zatem powiedzieć, że konstruktory są aktywowane w kolejności dziedziczenia — od klasy bazowej ze szczytu hierarchii, poprzez kolejne klasy w hierarchii aż do ostatniej klasy pochodnej.

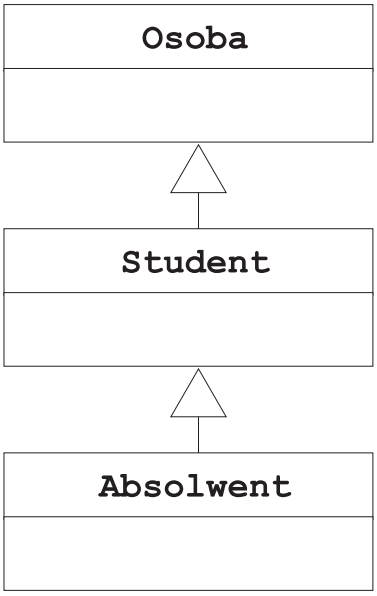
Klasy tworzące hierarchię mogą mieć zdefiniowane destruktory. W momencie usuwania obiektu klasy pochodnej zostanie automatycznie aktywowany jego destruktor, a następnie destruktor jego klasy bazowej. Jeżeli ta jest również klasą pochodną, następuje aktywowanie destruktora jej klasy bazowej, proces ten trwa tak długo, aż zostanie aktywowany destruktor klasy ze szczytu hierarchii klas.

Z aktywowaniem konstruktorów w hierarchii klas wiążą się pewne przypadki szczególne. Omówimy je na przykładzie. Załóżmy, że mamy trzy klasy o nazwach `Absolwent`, `Student`, `Osoba`. Klasa `Absolwent` jest klasą pochodną klasy `Student`, ta zaś jest pochodną klasy `Osoba`. Klasy tworzą zatem hierarchię przedstawioną na rys. 10.

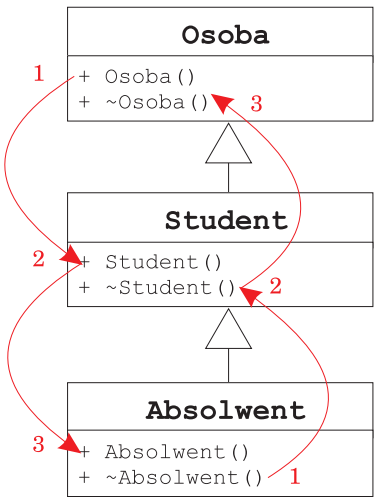
Zdefiniowane w tych klasach konstruktory będą uaktywniane zgodnie z porządkiem dziedziczenia — najpierw konstruktor klasy `Osoba`, potem klasy `Absolwent` a na końcu klasy `Student`, kolejność aktywowania destruktorów będzie odwrotna (kolejność i kierunek wywołań ilustruje rys. 11).

Załóżmy, że klasa bazowa `Osoba` posiada jedno pole, będące jej unikatowym numerem. Załóżmy również, że każda z klas posiada własny konstruktor domyślny i ogólny, a także destruktor. Niech dla potrzeb tego przykładu, każda z tych funkcji wyprowadza do strumienia wyjściowego informację, że to ona właśnie działa:

```
class Osoba {
  public:
    Osoba() : idOsoby( 0 ) {
      cout << "\nKonstruktor Osoba()";
    }
    Osoba( int id ) : idOsoby( id ) {
      cout << "\nKonstruktor Osoba( " << id << " )";
    }
    ~Osoba() {
      cout << "\nDestruktor ~Osoba()";
    }
    int idOsoby;
};
```



Rysunek 10: Przykładowa hierarchia klas



Rysunek 11: Aktywowanie konstruktorów i destruktorów

```
class Student : public Osoba {
  public:
    Student() : Osoba() {
      cout << "\nKonstruktor Student()";
    }
    Student( int id ) : Osoba( id ) {
      cout << "\nKonstruktor Student( " << id << " )";
    }
    ~Student() {
      cout << "\nDestruktor ~Student()";
    }
};

class Absolwent : public Student {
  public:
    Absolwent() : Student() {
      cout << "\nKonstruktor Absolwent()";
    }
    Absolwent( int id ) : Student( id ) {
      cout << "\nKonstruktor Absolwent( " << id << " )";
    }
    ~Absolwent() {
      cout << "\nDestruktor ~Absolwent()";
    }
};
```

Załóżmy, że w obrębie pewnego bloku zdefiniowano obiekt `a` klasy `Absolwent`:

```
. . .
{
  Absolwent a;
  . . .
}
. . .
```

Na etapie definiowania obiektu `a` zostanie uaktywniony jego konstruktor. Ponieważ na jego liście inicjalizacyjnej występuje uaktywnienie konstruktora jego klasy bazowej `Student`, to on powinien wykonać się wcześniej. Ale na jego liście inicjalizacyjnej występuje nazwa konstruktora klasy `Osoba`, zatem to on wykona się jako pierwszy — zgodnie z rysunkiem 11. W momencie zakończenia wykonania instrukcji blokowej, w obrębie której zdefiniowano obiekt `a`, obiekt ten będzie usuwany z pamięci operacyjnej. Spowoduje to aktywowanie jego destruktora, a potem destruktorów kolejnych klas bazowych, zgodnie z kolejnością przedstawioną na rys. 11. Wynik działania opisywanego programu prezentuje rys. 12 i rzeczywiście jest on zgodny z oczekiwaniami.

Podobnie będzie w przypadku definicji obiektu `a` z wykorzystaniem parametru inicjalizującego. Również nastąpi aktywowanie

```
Konstruktor Osoba()
Konstruktor Student()
Konstruktor Absolwent()
Destruktor ~Absolwent()
Destruktor ~Student()
Destruktor ~Osoba()_
```

Rysunek 12: Aktywacja konstruktorów domyślnych i destruktorów

konstruktorów — tym razem ogólnych — zgodnie z hierarchią dziedziczenia (co ilustruje rys. 13):

```
. . .
{
  Absolwent a( 10 );
  . . .
}
. . .
```

```
Konstruktor Osoba( 10 )
Konstruktor Student( 10 )
Konstruktor Absolwent( 10 )
Destruktor ~Absolwent()
Destruktor ~Student()
Destruktor ~Osoba()
```

Rysunek 13: Aktywacja konstruktorów ogólnych i destruktorów

Na tym etapie rozważań nieodparcie nasuwa się spostrzeżenie, że w aktywowanie konstruktorów nie ma nic tajemniczego — przecież każdy konstruktor aktywuje najpierw konstruktor klasy bazowej na swojej liście inicjalizacyjnej. Tak dzieje się kolejno, na coraz wyższych poziomach hierarhii klas. Zatem rzeczywiście, jako pierwszy powinien wykonać się konstruktor klasy `Osoba`, bo ten nie posiada klasy bazowej.

Zaskoczyć nieco może natomiast automatyzm aktywowania destruktorów — żaden z destruktorów nie wywołuje i nie aktywuje innego destruktora, są one wywoływane automatycznie przez kompilator. Okazuje się jednak, że taki automatyzm działa również dla konstruktorów. Przeprowadźmy następujący eksperyment — usuńmy aktywację konstruktorów klas bazowych z listy inicjalizacyjnej konstruktorów domyślnych klas `Absolwent` i `Student`. Po rekompilacji i uruchomieniu zmodyfikowanego programu, konstruktory będą aktywowane tak jak poprzednio, a wynik działania programu będzie dokładnie taki jak na rys. 12.

Kompilator zauważa, że konstruktory nie inicjują swojej kasy bazowej i postanawia zrobić to za nas, wywołując automatycznie konstruktory kolejnych klas bazowych. Dlaczego zatem w prezentowanych przykładach mozolnie aktywujemy konstruktory klas bazowych na liście inicjalizacyjnej? Bo tak powinno być — taka jest dobra praktyka programistyczna, nie pozostawiająca wątpliwości jakie są intencje programisty.

Uwaga — automatyzm wykorzystuje konstruktory domyślne. Jeżeli usuniemy aktywację konstruktora ogólnego klasy `Student` z listy inicjalizacyjnej konstruktora klasy `Absolwent`, to kompilator owszem, wywoła automatycznie konstruktory klas bazowych, ale *domyślne*! Wyniki działania tak zmodyfikowanego programu przykładowego prezentuje rysunek 14.

```
Konstruktor Osoba()
Konstruktor Student()
Konstruktor Absolwent( 10 )
Destruktor ~Absolwent()
Destruktor ~Student()
Destruktor ~Osoba()
```

Rysunek 14: Automatyczna aktywacja konstruktorów domyślnych

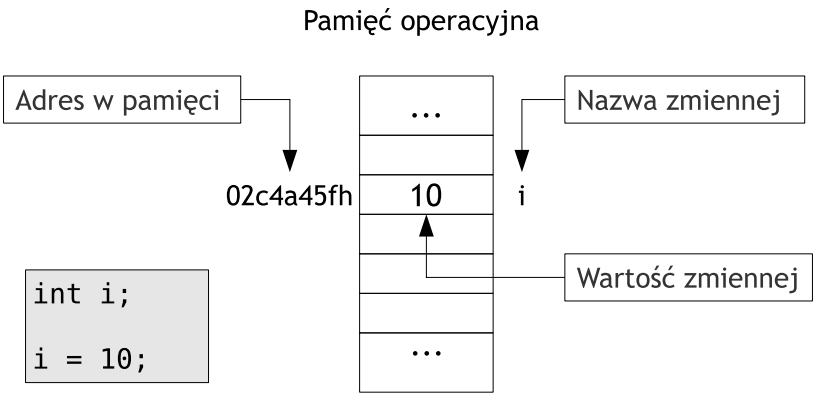
Jakie wnioski wypływają z rozważań przedstawionych w tym podrozdziale? Automatyzm aktywowania destruktorów jest korzystny i przewidywalny. Destruktor w danej klasie jest zawsze jeden, zatem nie ma wątpliwości, który zostanie wywołany. W przypadku konstruktorów jest inaczej — ich nazwy mogą być przeciążane, konstruktorów może być więcej. Dodatkowo w przypadku braku konstruktorów, kompilator syntetyzuje pusty, nie robiący nic pożytecznego konstruktor domyślny. To wszystko powoduje, że automatyczne aktywowanie konstruktorów może czasem być realizowane inaczej niż chcemy. Warto zatem nie pozostawiać kompilatorowi

swobody, definiować odpowiednie konstruktory[10] i aktywować na ich liście inicjalizacyjnej konstruktory klas bazowych.

[10] W omawianym przypadku można nie definiować w każdej z klas osobnego konstruktora domyślnego, a zastosować konstruktory ogólne z parametrem domyślnym o wartości 0 (jak w klasie *Kwadrat* na str. 26).

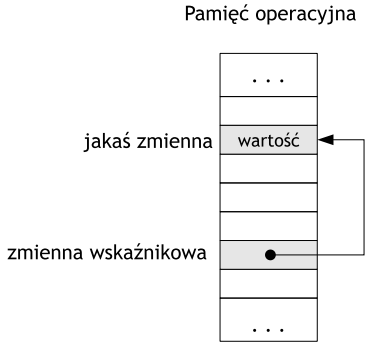
Zmienne wskaźnikowe, referencje i dynamiczny przydział pamięci

Niezależnie od przyjętej metody konstruowania programów, *zmienna* jest elementem programu rezydującym w pamięci operacyjnej, przeznaczonym do przechowywania wartości pewnego typu. Zmienna jest podstawowym elementem przechowującym dane, na których program operuje. Każda zmienna posiada swoją *nazwę* oraz *typ wartości*, jest przechowywana w pamięci operacyjnej, liczba zajętych bajtów zależy właśnie od typu zmiennej. Nazwa zmiennej *identyfikuje* zmienną w programie, zwalniając programistę od zastanawiania się, pod jakim adresem w pamięci zmienna jest zlokalizowana. Ilustruje to rys. 15.



Rysunek 15: Zmienna jako obiekt w pamięci operacyjnej

W języku C++ intensywnie wykorzystuje się pewien szczególny rodzaj zmiennych — *zmienne wskaźnikowe* oraz wyrażenia te zmienne wykorzystujące. Dokładne opanowanie zasad posługiwania się wskaźnikami jest niezbędne do efektywnego i sprawnego programowania w języku C++. Tej umiejętności nie można pominąć, przeskoczyć lub zostawić na później. Koncepcja zmiennych wskaźnikowych oraz metody ich wykorzystania są proste, wymagają one jednak uwagi, zrozumienia i myślenia.



Rysunek 16: Koncepcja zmiennej wskaźnikowej

Zmienne wskaźnikowe

Zmienna wskaźnikowa przeznaczona jest do lokalizowania (inaczej wskazywania) obiektów w pamięci operacyjnej. Jedyną rolą zmiennej wskaźnikowej jest umożliwienie odwoływania się do obiektów wskazywanych. Obiektami wskazywanymi mogą być inne zmienne, funkcje oraz nienazwane obszary pamięci operacyjnej — zwykle przydzielane programowi dynamicznie, w trakcie jego działania (co zostanie omówione w dalszej części tego rozdziału). Zatem zmienna wskaźnikowa:

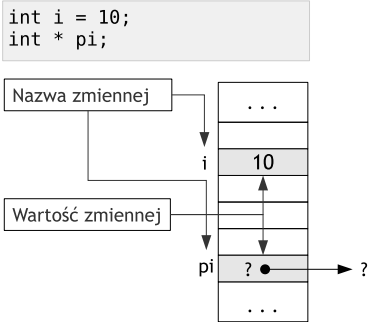
- sama rezyduje w pamięci operacyjnej;

- wobec powyższego, sama może być również „wskazywana” przez inną zmienną wskaźnikową;
- służy do lokalizowania w pamięci operacyjnej innych zmiennych, nienazwanych bloków pamięci oraz bloków zawierających kod programu, np. funkcji.

Koncepcję zmiennej wskaźnikowej ilustruje rys. 16. Zwykle przyjmuje się, że zmienna wskaźnikowa zawiera w sobie adres obiektu wskazywanego, jednak nie musi ona w sobie zawierać adresu bezpośredniego (fizycznego) a inną informację, pozwalającą na precyzyjne i jednoznaczne zidentyfikowanie położenia obiektu w pamięci (np. tzw. *offset* w przypadku procesorów z rodziny Intel8086).

Jak definiuje się zmienne wskaźnikowe i jak się je wykorzystuje? Rozważmy następujący przykład:

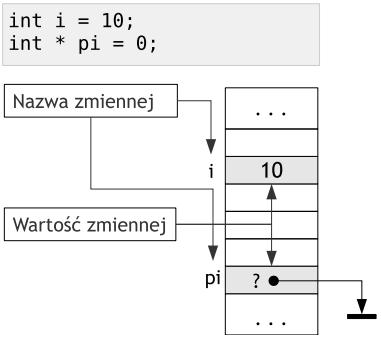
```
int i = 10;
int * pi;
```



Rysunek 17: Deklaracja zmiennej wskaźnikowej

Deklaracja pierwszej zmiennej nie budzi wątpliwości — `i` to „zwykła” zmienna typu całkowitoliczbowego, zainicjowana wartością 10. W deklaracji zmiennej `pi` występuje nowy symbol ‘*’. Oznacza on, że zmienna `pi` jest *zmienną wskaźnikową*, przeznaczoną do wskazywania w pamięci operacyjnej obiektów typu `int`. Zawartość pamięci operacyjnej odpowiadającą powyższym deklaracjom prezentuje rys. 17.

Zwróćmy uwagę, że tuż po deklaracji, zmienna wskaźnikowa ma wartość zależną od zasięgu w jakim została zdefiniowana. Jeżeli zmienna `pi` jest *automatyczną*, jej wartość jest przypadkowa. Dla zmiennej wskaźnikowej oznacza to, że wskazuje ona na przypadkowy obiekt w pamięci operacyjnej. Aby tak nie było, zwykle zmienne wskaźnikowe zeruje[11] się na etapie ich deklaracji. Tak zainicjowana zmienna wskaźnikowa nie wskazuje na żaden obiekt. Ilustruje to „uziemnienie” strzałki symbolizującej wskaźnik na rys. 18.



Rysunek 18: Deklaracja wyzerowanej zmiennej wskaźnikowej

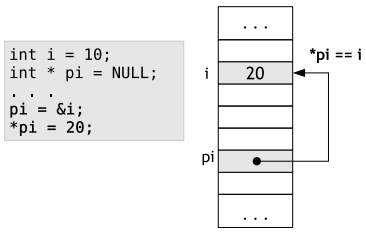
[11] W języku C wykorzystuje się symbol NULL jako wartość wskaźnika pustego — a więc, takiego który nie wskazuje niczego. W języku C++ zrezygnowano z wykorzystania tego symbolu, wykorzystuje się po prostu wartość 0.

Potrafimy już zatem zadeklarować zmienną wskaźnikową, zainicjować ją wartością oznaczającą brak powiązania z jakimkolwiek obiektem wskazywanym. W jaki sposób można powiązać taką zmienną z jakimś obiektem w pamięci? Przyjrzyjmy się bliżej następującym dwóm liniom kodu:

```
pi = &i;
*pi = 20;
```

W pierwszej linii, po lewej stronie operatora przypisania występuje nazwa zmiennej wskaźnikowej `pi`, zatem do do tej zmiennej zostania przypisana wartość występująca po prawej stronie przypisania. Co oznacza zapis `&i`? Jednoargumentowy operator `&` buduje wyrażenie wskaźnikowe lokalizujące zmienną `i` w pamięci operacyjnej. Zatem do zmiennej wskaźnikowej `pi` wstawiany jest wskaźnik do zmiennej `i`. Od tego momentu do obszaru pamięci zajmowanego przez zmienną `i` możemy się odwoływać wykorzystując jej nazwę lub za pośrednictwem wskaźnika zapisanego w zmiennej `pi`.

Takie właśnie odwołanie zawiera druga linia rozważanego przykładu: `*pi= 20`. Zapis `*pi` oznacza obiekt wskazywany przez `pi`, może on wystąpić wszędzie tam, gdzie może wystąpić `i`. Symbol `*` oznacza jednoargumentowy operator adresowania pośredniego, którego wynikiem jest obiekt wskazywany przez wskaźnik zapisany w zmiennej `pi`. Zatem powyższe przypisanie spowoduje wstawienie wartości 20 do zmiennej wskazywanej przez wskaźnik `pi`, a zatem do zmiennej `i`. Ilustruje to rys. 19.



Rysunek 19: Wykorzystanie zmiennej wskaźnikowej

Zmienne wskaźnikowe mają szereg zastosowań, zwykle dzieli się je na następujące kategorie:

- realizacja przekazywania parametrów przez zmienną;
- wykorzystanie pamięci zarządzanej dynamicznie;
- manipulowanie tablicami;
- budowa rekurencyjnych struktur danych.

W tym opracowaniu zostaną przedstawione tylko wybrane zagadnienia związane z zastosowaniem zmiennych wskaźnikowych, dokładny opis zastosowań pozostałych znaleźć można w literaturze.

Zmienne wskaźnikowe a przekazywanie parametrów

W językach C i C++ przekazywanie parametrów do funkcji odbywa się *przez wartość*. Oznacza to, że parametr *aktualny* wywołania kopiowany jest do parametru *formalnego* funkcji. Wnętrze funkcji operuje na kopii wartości orginalnej, nie mogąc zmienić oryginalnej wartości parametru aktualnego wywołania funkcji.

W przedstawionym niżej przykładzie, parametrem aktualnym wywołania jest zmienna `a`. Jej wartość — a zatem liczba 5 — kopiowana jest do parametru formalnego funkcji `inc` o nazwie `i`. Wnętrze funkcji `inc` zwiększa wartość parametru `i` o jeden, operacja ta dotyczy jednak dokładanie tego parametru, wartość zmiennej `a` nie jest modyfikowana. Innymi słowy — operacja zwiększenia o jeden dotyczy, skopiowanej do parametru formalnego `i`, wartości 5. Zatem ten fragment programu wyprowadzi do strumienia wyjściowego niezmodyfikowaną wartość zmiennej `a` równą 5.

```
void inc( int i ) {
  i = i + 1;
}
. . .
int a = 5;
inc( a );
cout << a;
```

Aby funkcja `inc` mogła zwiększyć wartość zmiennej `a`, należy przekazać jej wskaźnik na tą zmienną. Wtedy wnętrze funkcji może odwołać się do oryginalnej wartości poprzez odwołanie pośrednie zrealizowane z wykorzystaniem symbolu `*`. Ilustruje to przedstawiony niżej, zmodyfikowany fragment programu. Zauważmy, że

parametrem aktualnym wywołania funkcji jest `&a`, zatem funkcji przekazujemy wskaźnik do zmiennej `a`. Wskaźnik ten kopiowany jest do parametru `i`, który jest zmienną wskaźnikową. Zwiększenie o jeden dotyczy obiektu wskazywanego przez wskaźnik `i`, a tym obiektem jest właśnie zmienna `a`.

```
void inc( int * i ) {
  *i = *i + 1;
}
. . .
int a = 5;
inc( &a );
cout << a;
```

Ten mechanizm przekazywania wartości naśladuje spotykane np. w języku Pascal *przekazywanie parametrów przez zmienną*. Jednak wykorzystanie wskaźników jest w tym przypadku przez wielu uznawane za niewygodne, w języku C++ można skorzystać z parametrów będących *referencjami*.

Zmienne referencyjne

Omawianą powyżej funkcję `inc` można zrealizować bez wykorzystania wskaźników. W języku C++ wprowadzono zmienne referencyjne. Takie zmienne mogą „nakładać się" na inne zmienne, stanowiąc ich alternatywną nazwę. Zmienna referencyjna może być rozumiana jako *alias*, *alternatywna nazwa* jakiegoś obszaru pamięci. Dobrym przykładem jest właśnie referencyjna wersja funkcji `inc`:

```
void inc( int & i ) {
  ++i;
}
. . .
int a = 5;
inc( a );
cout << a;
```

W tym przykładzie zmienną o charakterze referencyjnym jest parametr formalny funkcji `inc`. W jego deklaracji występuję symbol `&`, oznaczający, że parametr `i` będzie nakładał się na obszar pamięci parametru aktualnego, z jakim zostanie wywołana funkcja `inc`. Rzeczywiście, jeżeli wywołamy tę funkcję z parametrem aktualnym `a`, parametr formalny `i` „nałoży" się na zmienną `a` i każda modyfikacja `i` będzie automatycznie dotyczyła zmiennej `a`.

Mechanizm referencji w języku C++ pozwala na unikanie stosowania zmiennych wskaźnikowych w niektórych sytuacjach. Referencje są trochę tak jak wskaźniki, których nie trzeba jawnie kotwiczyć o inne obiekty (operator `&`) i jawnie dokonywać ich dereferencji (operator `*`). Zmienne referencyjnie mogą występować nie tylko jako parametry formalne funkcji. Rozważmy następujący przykład:

```
int i = 10;
int & ri = i;
ri = 20;
```

Zmienna `ri` jest zmienną referencyjną, może być aliasem każdego obiektu typu `int`. Na etapie deklarowania, zmienną referencyjną *należy* zainicjować, jednak przypisanie występujące w jej deklaracji *nie dotyczy* wartości zmiennych — zapis `int & ri = i` oznacza, że zmienna `ri` odnosiła się do tej samej lokalizacji w pamięci, którą zajmuje zmienna `i`. Inaczej mówiąc — zmienna `ri` zostanie „nałożona" na zmienną `i`. Jak należy się spodziewać, każda modyfikacja zmiennej `ri` jest jednocześnie modyfikacją zmiennej `i`.

Zmienne wskaźnikowe a obiekty

Czasem programista chce lub musi posługiwać się obiektem pewnej klasy nie za pośrednictwem jego nazwy, a za pośrednictwem zmiennych wskaźnikowych. Załóżmy, że wewnątrz pewnej funkcji `wypiszDane` musimy się odwołać do obiektu klasy `Kwadrat` (zobacz str. 29), przekazanego tej funkcji poprzez wskaźnik:

```
Kwadrat kwadr( 5 );
. . .
wypiszDane( &kwadr );
. . .
. . .
void wypiszDane( Kwadrat * k ) {
  cout << "\nDługość boku : " << (*k).podajDlBoku();
  cout << "\nPole kwadratu: " << (*k).obliczPole();
}
```

Parametr formalny funkcji `wypiszDane` jest wskaźnikiem, aby odwołać się do obiektu wskazywanego, należy dokonać dereferencji wskaźnika: `*k`. Po ujęciu takiego wyrażenia w nawiasy: `(*k)` reprezentuje już ono obiekt `kwadr`, do którego wskazanie było parametrem wywołania funkcji. Zatem po wyrażeniu `(*k)` możemy postawić kropkę i odwołać się do funkcji składowych obiektu klasy `Kwadrat`. Należy zwrócić uwagę na konieczność stosowania nawiasów — priorytety operatorów '.' i '*' są tak ustalone, że zapis: `*k.obliczPole()` oznaczałby, że `k` jest obiektem, a `obliczPole()` to wskaźnik, którego dereferencji chcemy dokonać za pośrednictwem operatora '*'.

Taki zapis odwołań wskaźnikowych do obiektów jest niewygodny. Ponieważ występuje on jednak w języku C++ bardzo często, wprowadzono specjalną notację dla wskaźników odwołujących się do rekordów (struktur) i obiektów. Zamiast pisać `(*k).obliczPole()` możemy napisać: `k->obliczPole()`

Złożenie znaku '-' i '>' daje operator odwołania się do składowej rekordu lub obiektu, co pozwala np. zapisać funkcję `wypiszDane` w następujący sposób:

```
void wypiszDane( Kwadrat * k ) {
  cout << "\nDługość boku : " << k->podajDlBoku();
  cout << "\nPole kwadratu: " << k->obliczPole();
}
```

Dynamiczny przydział pamięci

Dynamiczny przydział pamięci polega na rezerwowaniu fragmentu pamięci w obszarze tzw. *pamięci wolnej*. Programista decyduje o tym kiedy zarezerwuje blok pamięci, jaki on będzie miał rozmiar oraz kiedy zrezygnuje z jego wykorzystania, zwracając zarezerwowaną pamięć do puli obszarów wolnych. Pamięć wolna nazywana jest potocznie *stertą* (ang. heap); ten obszar pamięci jest:

- przeznaczony do przechowywania danych dynamicznych,
- kontrolowany ręcznie przez programistę,
- ograniczony pod względem rozmiaru,
- przydzielany pasującymi fragmentami.

Typowy scenariusz wykorzystania dynamicznego przydziału pamięci wygląda następująco:

- określenie wielkości potrzebnego obszaru pamięci;
- przydział pamięci i zapamiętanie wskazania tego obszaru w zmiennej wskaźnikowej;
- sprawdzenie czy przydział pamięci się powiódł, jeżeli tak to:
  - wykorzystanie przydzielonego bloku pamięci;
  - zwolnienie przydzielonego bloku pamięci, gdy nie jest już potrzebny.

W języku C++ można używać znanych z C funkcji `malloc`, `calloc`, ... Jednak w C++wprowadzono specjalne operatory zarządzające pamięcią: `new` oraz `delete`. Ich wykorzystanie jest zalecane, a w wielu przypadkach konieczne. Operatory są częścią języka i współpracują z mechanizmami kontroli typów. Zatem w języku C++ zarządzanie pamięcią dynamiczną realizować będzie:

- operator `new` — przydział pamięci (wraz z aktywowaniem odpowiedniego konstruktora),
- operator `delete` — zwolnienie pamięci (wraz z aktywowaniem destruktora, jeżeli go zdefiniowano).

Scenariusz przydziału pamięci dla obiektu klasy `Kwadrat` zawiera przedstawiony niżej przykład:

```
Kwadrat * kwadr;
kwadr = new Kwadrat( 5 );
if( kwadr != 0 ) {
```

```
  cout << "\nDługość boku : " << kwadr->podajDlBoku();
  cout << "\nPole kwadratu: " << kwadr->obliczPole();
  delete kwadr;
}
```

Instrukcja `kwadr = new Kwadrat( 5 )` przydziela pamięć dla obiektu klasy `Kwadrat`, inicjując go przy okazji aktywacją konstruktora ogólnego, z parametrem o wartości 5. Rezultatem działania operatora `new` jest wskaźnik na nowo utworzony obiekt, wskaźnik ten zapamiętywany jest w zmiennej `kwadr`. Instrukcja warunkowa testuje, czy wartość wskaźnika jest niezerowa — a więc, czy przydział pamięci się powiódł. Jeżeli tak jest, wykonujemy operacje na obiekcie wskazywanym przez zmienną `kwadr` a potem zwalniamy pamięć operatorem `delete`.

Ten schemat zakłada, że rezultatem operatora `new` będzie wartość 0 jeżeli przydział pamięci się nie powiedzie. Tak rzeczywiście było w starszych wersjach języka C++. Zgodnie z aktualnie obowiązującym standardem, aby uzyskać takie zachowanie operatora `new`, należy posłużyć się jego specjalną wersją i dokonać przydziału pamięci w następujący sposób:

```
. . .
kwadr = new (nothrow) Kwadrat( 5 );
. . .
```

Jeżeli nie posłużymy się specyfikacją `(nothrow)`, kompilator użyje wersji operatora `new`, która wygeneruje *wyjątek* `bad_alloc` oznaczający zazwyczaj brak pamięci dla nowego obiektu. Wyjątek taki należy obsłużyć używając instrukcji `try-catch`, co ilustruje podany niżej fragment kodu.

```
Kwadrat * kwadr;
try {
  kwadr = new Kwadrat( 5 );
  cout << "\nDługość boku : " << kwadr->podajDlBoku();
  cout << "\nPole kwadratu: " << kwadr->obliczPole();
  delete kwadr;
}
catch( bad_alloc ) {
   cout << "Brak pamięci dla programu!";
}
```

Polimorfizm i funkcje wirtualne

Dziedziczenie jest jednym z najistotniejszych mechanizmów programowania obiektowego. Dzięki niemu, nie trzeba budować nowych klas od podstaw, można bowiem wykorzystywać istniejące klasy jako bazę dla nowych klas. Nowe klasy budujemy rozszerzając klasy bazowe — dodając nowe dane i/lub funkcje składowe.

Stosowanie dziedziczenia prowadzi do powstania hierarchii klas — taką hierarchię można przedstawić graficznie, tak jak na rysunkach 8, 9, 10. Przyjrzyjmy się bliżej hierarchii klas przedstawionej na

rys. 10. Obiekty klas `Osoba`, `Student`, `Absolwent` są „spokrewnione" — ich klasy powiązane są związkiem dziedziczenia. Owo spokrewnienie oznacza również, że obiekty tych klas są do siebie *podobne*. Rzeczywiście, obiekt klasy `Student` *jest* jednocześnie obiektem klasy `Osoba`, obiekt klasy `Absolwent` *jest* jednocześnie obiektem klasy `Student`, ale również obiektem klasy `Osoba`. Dochodzimy w tym momencie do do bardzo istotnego spostrzeżenia — obiekty klas budujących hierarchię dziedziczenia mogą występować w różnych postaciach. Ta cecha programowania obiektowego nazywana jest *polimorfizmem* i ma wpływ na konkretne rozwiązania w programowaniu obiektowym. Słowo polimorfizm pochodzi od dwóch greckich słów *poly* czyli *wiele* oraz *morphos* czyli *postać* i oznacza *wielopostaciowość*. Oznacza to, że obiekt może przyjąć jedną z wielu dozwolonych dla niego postaci i wykazywać się innym zachowaniem w zależności od tego, w jakiej postaci występuje. Co to oznacza w praktyce i jak z tego skorzystać? Odpowiedź można znaleźć w dalszej części tego rozdziału.

Obiekty, wskaźniki, referencje a hierarchia klas

Załóżmy, że kod odpowiadający hierarchii klas przedstawionej na rys. 10 ma następującą postać[12]:

[12] Realizacje klas zostały zmienione w stosunku do podanych wcześniej, istota rozważań bowiem jest inna.

```
class Osoba {
  public:
    Osoba() : nrDowodu( 0 )
    {}
    void piszKimJestes() {
      cout << "Osoba\n";
    }
    int nrDowodu;
};

class Student : public Osoba {
  public:
    Student() : Osoba(), nrIndeksu( 0 )
    {}
    void piszKimJestes() {
      cout << "Student\n";
    }
    int nrIndeksu;
};

class Absolwent : public Student {
  public:
    Absolwent() : Student(), nrDyplomu( 0 )
    {}
    void piszKimJestes() {
      cout << "Absolwent\n";
    }
```

```
    int nrDyplomu;
};
```

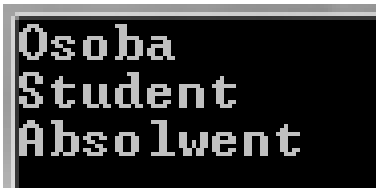
W klasie `Osoba` zdefiniowano pole `nrDowodu`[13], w klasie `Student` pole `nrIndeksu` a w klasie `Absolwent` pole `nrDyplomu`. Dla potrzeb klarowności tego przykładu pola te pozostawiamy publicznymi[14]. Każda z klas dokonuje redefinicji funkcji składowej `piszKimJestes`, tak aby do strumienia wyjściowego programu została skierowana nazwa odpowiedniej klasy. Rozważmy następujący przykład:

[13] W rzeczywistości pole opisujące numer dowodu osobistego nie powinno być typu całkowitoliczbowego. Numer dowodu zawiera zwykle symbole literowe. Zastosowano typ całkowity dla uproszczenia rozważań.

[14] Zgodnie z zasadą hermetyzacji powinniśmy te pola uprywatnić i zdefiniować dla nich akcesory

```
Osoba o;
Student s;
Absolwent a;
o.piszKimJestes();
s.piszKimJestes();
a.piszKimJestes();
```

Wynikiem jego działania będą informacje wyprowadzane do strumienia wyjściowego programu przez każdą z funkcji, zgodnie z klasą do której obiekt należy — ilustruje to rys. 20.



Rysunek 20: Wyniki działania funkcji *piszKimJestes*

Klasa `Student` jest klasą pochodną klasy `Osoba`, zatem każdy obiekt klasy `Student` jest jednocześnie obiektem klasy `Osoba`. Zatem następujące przypisanie: `o = s;` jest prawidłowe. Można przypisać do obiektu klasy bazowej obiekt klasy pochodnej. Jednak przy takim przypisaniu dokonywana jest konwersja typów, tracimy dostęp do składowych zdefiniowanych w klasie pochodnej, odwołanie: `o.nrIndeksu = 121212;` jest nieprawidłowe, bowiem obiekt `o` jest przecież reprezentantem klasy `Osoba`, a ta nie posiada pola `o.nrIndeksu`. Nieprawidłowe jest również przypisanie odwrotne: `s = o;` Gdyby było ono dozwolone, można by następnie próbować odwoływać się do `nrIndeksu` obiektu `s` — jego wartość byłaby nieokreślona, bowiem takiego pola nie ma w obiekcie `o`.

W podobny sposób możemy realizować odwołania za pośrednictwem wskaźników i referencji. Jeżeli istnieje wskaźnik do klasy bazowej, wolno nam do niego przypisać wskazanie na obiekt tejże klasy, jak również wskazanie na obiekt klasy pochodnej:

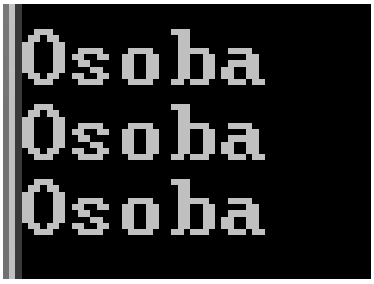
```
Osoba * wo;
wo = &o;
. . .
wo = &a;
. . .
wo = &s;
. . .
Osoba & r1 = o;
Osoba & r2 = s;
Osoba & r3 = a;
. . .
```

Wielopostaciowość a hierarchia klas

Skoro takie przypisania są możliwe, zastanówmy się, jaki będzie wynik działania następującego fragmentu kodu:

```
Osoba o;
Student s;
Absolwent a;
o.piszKimJestes();
o = s;
o.piszKimJestes();
o = a;
o.piszKimJestes();
```

Intuicja pozwala się spodziewać, że otrzymamy wyniki podobne do przedstawionych na rys. 20 — po przypisaniu `o = s` obiekt `o` przyjmuje przecież postać obiektu klasy `Student`, a potem w wyniku przypisania `o = a` postać obiektu klasy `Absolwent`. Niestety, wynik działania przedstawionego wyżej prezentuje rys. 21.



Rysunek 21: Wywołanie funkcji *piszKimJestes* dla obiektu *o*

Intuicja zawiodła — obiekt `o` zachowuje się jak obiekt klasy `Osoba`, mimo iż przypisaniem zasugerowaliśmy mu, żeby przyjął postać najpierw obiektu `s` klasy pochodnej `Student` a potem obiektu `a` klas pochodnej `Absolwent`. Okazuje się, że wspomniany wcześniej polimorfizm nie działa — powyższe przypisania *przekształcają* obiekty klas pochodnych w obiekt klasy bazowej, a informacja o ich różnorodności ginie.

Sprawdźmy, czy podobnie będzie w przypadku wykorzystania zmiennych wskaźnikowych, rozważmy następujący przykład:

```
Osoba * wo;
wo = &o;
wo->piszKimJestes();
wo = &s;
wo->piszKimJestes();
wo = &a;
wo->piszKimJestes();
```

Okazuje się, że nic się nie zmienia — wyniki działania powyższego kodu są analogiczne do wcześniejszych, przedstawionych przez rys. 21. Mimo zakotwiczenia wskaźnika `wo` do obiektu klasy `Student`, wywołana zostanie funkcja `piszKimJestes` zgodnie z typem występującym w deklaracji tego wskaźnika.

Dlaczego tak jest? Zastanówmy się co robi kompilator w momencie natrafienia na linię w postaci: `wo->piszKimJestes();` Kompilator zauważa, że następuje wywołanie funkcji składowej `piszKimJestes`, odnajduje ją sprawdzając typ wskaźnika, za pośrednictwem którego tę funkcję wywołujemy. W efekcie kompilator wstawia w tym miejscu kod wywołujący funkcję `Osoba::piszKimJestes`. Tak wywoływane są wszystkie funkcje i to niezależnie, czy programujemy obiektowo czy nie. Zauważmy, że to, jak funkcja zostanie wywołana

rozstrzyga się już na etapie kompilacji, taki rodzaj wywołania nazywa się *wywołaniem statycznym*. Inaczej mówimy, że następuje *wczesne wiązanie* wywołania z konkretną wersją wywoływanej funkcji.

Późne wiązanie i funkcje wirtualne

Kompilator musiałby działać zupełnie inaczej, gdyby miał wywołać funkcję zgodne *z klasą obiektu na który wskazuje* `wo` — a więc funkcję z klasy `Student` lub klasy `Absolwent`. Zadanie nie jest proste — kompilator musiałby śledzić na jaki obiekt wskazuje `wo`. Proces ten mógłby być złożony, a w niektórych przypadkach zupełnie nieskuteczny — często na etapie kompilacji nie wiemy jeszcze na na jaki obiekt bedzie rzeczywiście wskazywała zmienna wskaźnikowa. Trudno jest zatem związać wywołania funkcji z jej konkretną wersją *statycznie*. Czy w ogóle można zmusić kompilator, aby wywoływał funkcję zgodnie z klasą obiektu wskazywanego aktualnie przez wskaźnik? Okazuje się, że tak.

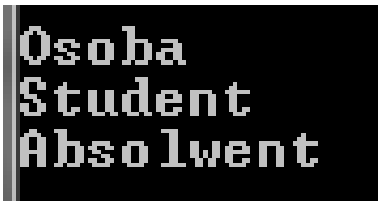
Wraz z programowaniem obiektowym wprowadzono do języków programowania tzw. *wiązanie dynamiczne*[15]. Polega ono na tym, że dopiero na etapie wykonania programu rozstrzyga się, która wersja funkcji składowej ma zostać wywołana. Taki rodzaj wiązania wywołania z określeniem funkcji wywoływanej nazywa się również *wiązaniem opóźnionym*.

[15] Wiązanie dynamiczne było oczywiście wykorzystywane w językach nieobiektowych, jednak była to raczej technika programowania (ang. callback functions) a nie możliwość syntaktycznie wbudowana w język programowania.

Aby kompilator wywoływał funkcję `piszKimJestes` zgodne *z klasą obiektu na który wskazuje* `wo`, musimy poinformować o tym kompilator języka C++. Owo poinformowanie polega na umieszczeniu słowa kluczowego `virtual` przed deklaracją funkcji, która ma być wiązana dynamicznie. W naszym przypadku wystarczy umieścić słowo `virtual` w deklaracji funkcji `piszKimJestes` w klasie `Osoba`:

```
class Osoba {
  public:
    . . .
    virtual void piszKimJestes() {
      cout << "Osoba\n";
    }
    . . .
};
```

Od tego momentu, we *wszystkich klasach pochodnych* funkcja `piszKimJestes` będzie podlegała wiązaniu dynamicznemu. W klasach pochodnych nie trzeba już używać słowa `virtual`. W efekcie wynik działania przykładu ze strony 48 będzie taki jak na rys. 22. Funkcje zadeklarowane z wykorzystaniem słowa kluczowego `virtual` nazywamy *funkcjami wirtualnymi*.



Rysunek 22: Wywołanie funkcji wirtualnej *piszKimJestes*

O tym, która wersja *funkcji wirtualnej* zostanie wywołana decyduje *typ obiektu* wskazywanego przez wskaźnik, lub *typ obiektu* do którego odnosi się referencja. Ta informacja ustalana jest na etapie wykonania programu, na podstawie dodatkowej informacji zapisanej w każdym obiekcie — *tablicy funkcji wirtualnych*. Szczegółowe

informacje na temat koncepcji i realizacji tej tablicy można znaleźć w literaturze, dla przykładów prezentowanych w tym opracowaniu szczegóły te nie są istotne. Każda funkcja zadeklarowana bez słowa kluczowego `virtual` będzie podlegała *wiązaniu statycznemu*, a więc wersja funkcji do wywołania ustalana będzie na podstawie typu wskaźnika lub zmiennej referencyjnej.

W języku C++ programista może zatem decydować, czy funkcja składowa będzie wiązana statycznie czy dynamicznie. Wywołania funkcji wiązanych statycznie są szybsze — w miejscu wywołania funkcji jej adres jest znany. Wywołania funkcji wiązanych dynamicznie będą wolniejsze — w miejscu wywołania należy najpierw odnaleźć adres właściwej funkcji, posługując się wspomnianą wcześniej tablicą funkcji wirtualnych. Nasuwa się pytanie — czy narzut czasu związany z wywołaniem funkcji wirtualnej jest znaczący? Odpowiedź nie jest jednoznaczna, zależy to bowiem od efektywności sprzętu na jakim działa program. Jednak w przypadku komputera osobistego w typowej, nowoczesnej lecz niewyrafinowanej konfiguracji, różnica czasu wywołania funkcji niewirtualnej i wirtualnej jest pomijalnie mała. Nie ma zatem istotnych powodów aby unikać funkcji wirtualnych z powodów wydajnościowych. Pojawia się jednak kolejne pytanie — jakie są korzyści stosowania metod wirtualnych? Odpowiedzi na to pytanie ma dostarczyć kolejny podrozdział.

Funkcje wirtualne w akcji — hierarchia klas

Aby przedstawić konkretne przykłady wykorzystania funkcji wirtualnych, rozwiniemy nieco wprowadzone wcześniej definicje klas `Osoba`, `Student` oraz `Absolwent`. Klasę `Osoba` uzupełnimy o pola przeznaczone do przechowywania imienia i nazwiska — dla uproszczenia pozostawimy pola publicznymi i wykorzystamy zdefiniowany w bibliotece standardowej typ `string`:

```
class Osoba {
  public:
    Osoba()
      : imie( "" ), nazwisko( "" ), nrDowodu( 0 )
    {}
    Osoba( string i, string n, int nr )
      : imie( i ), nazwisko( n ), nrDowodu( nr )
    {}
    virtual void piszKimJestes() {
      cout << "\nOsoba\n";
    }
    virtual void wypiszDane() {
      piszKimJestes();
      cout << imie << " " << nazwisko << endl;
      cout << "Nr  dowodu: " << nrDowodu << endl;
    }
    int    nrDowodu;
    string imie;
```

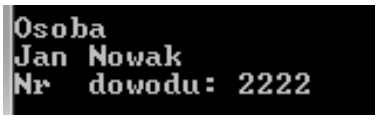
```
    string nazwisko;
};
```

Konstruktor bezparametrowy klasy `Osoba` inicjuje obiekt wartościami domyślnymi — są to odpowiednio napisy puste dla pól przechowujących imię i nazwisko, oraz wartość 0 dla pola przechowującego numer dowodu. Konstruktor ogólny otrzymuje trzy parametry — imię przekazane parametrem `i`, nazwisko przekazane parametrem `n` oraz numer dowodu przekazany parametrem `nr`. Wartości tych parametrów są kopiowane na liście inicjalizacyjnej do odpowiednich pól obiektu klasy `Osoba`. Ciało konstruktora może pozostać puste. Klasa została wyposażona w nową funkcję składową `wypiszDane`, która wyprowadza do strumienia wyjściowego informacje o klasie obiektu (wywołanie funkcji składowej `piszKimJestes`) oraz zawartości pól przechowujących imię, nazwisko i numer dowodu. Zapamiętajmy, że obie funkcje składowe są funkcjami wirtualnymi.

Obiekty klasy `Osoba` możemy tworzyć dynamicznie w następujący sposób:

```
Osoba * wo = new Osoba( "Jan", "Nowak", 2222 );
. . .
wo->wypiszDane();
. . .
delete wo;
```

Obiekt tworzony operatorem `new` otrzymuje komplet danych za pośrednictwem parametrów konstruktora ogólnego. Funkcja składowa `wypiszDane` wyprowadza je do strumienia wyjściowego w postaci przedstawionej na rys. 23.



Rysunek 23: Wynik działania funkcji *wypiszDane* klasy *Osoba*

Na podstawie klasy `Osoba` budujemy klasę pochodną `Student`, posiadającą dodatkowe pole przechowujące informację o numerze indeksu. W klasie tej zdefiniowano konstruktor bezparametrowy, inicjujący obiekt wartościami domyślnymi oraz konstruktor posiadający cztery parametry — trzy pierwsze są analogiczne do otrzymywanych przez konstruktor parametrowy klasy `Osoba`, czwarty parametr inicjalizuje pole `nrIndeksu`. Zwróćmy uwagę, że do zainicjowania pól odziedziczonych po klasie `Osoba` wykorzystywany jest konstruktor tejże klasy, aktywowany na liście inicjalizacyjnej konstruktora klasy `Student`. Również na liście inicjalizacyjnej otrzymuje swoją wartość pole `nrIndeksu`. Definicja klasy `Student` ma następująca postać:

```
class Student : public Osoba {
  public:
    Student() : Osoba(), nrIndeksu( 0 )
    {}
    Student( string i, string n, int nr, int nrI )
      : Osoba( i, n, nr ), nrIndeksu( nrI )
    {}
    void piszKimJestes() {
```

```
      cout << "\nStudent\n";
    }
    void wypiszDane() {
      Osoba::wypiszDane();
      cout << "Nr indeksu: " << nrIndeksu << endl;
    }
    int nrIndeksu;
};
```

W klasie `Student` dokonano redefinicji funkcji składowych `piszKimJesteś` oraz `wypiszDane`. Pierwszy przypadek został omówiony wcześniej. Drugą funkcję rozpoczyna wywołanie zdefiniowanej w klasie `Osoba` funkcji `wypiszDane`, której zadaniem jest wyprowadzenie danych właściwych dla tej klasy — imienia, nazwiska, numeru dowodu, po czym następuję wyprowadzenia do strumienia wyjściowego numeru indeksu, a więc informacji właściwej dla klasy `Student`.

W podobny sposób zbudujemy klasę `Absolwent`. Klasa ta powstanie na bazie klasy `Student`, zestaw jej pól zostanie rozszerzony o pole przeznaczone do przechowywania numeru dyplomu. Również w tej klasie dokonana zostanie redefinicja funkcji składowych `piszKimJestes` oraz `wypiszDane`. Ta ostatnia rozpocznie swoje działanie od wywołania wersji zdefiniowanej w klasie `Student`, po czym wyprowadzi do strumienia wyjściowego informację o numerze dyplomu. Deklaracja klasy `Absolwent` ma następująca postać:

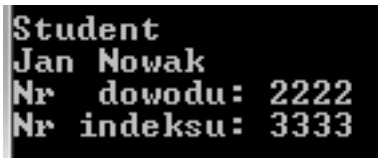
```
class Absolwent : public Student {
  public:
    Absolwent() : Student(), nrDyplomu( 0 )
    {}
    Absolwent( string i, string n, int nr, int nrI, int nrD )
      : Student( i, n, nr, nrI ), nrDyplomu( nrD )
    {}
    void piszKimJestes() {
      cout << "\nAbsolwent\n";
    }
    void wypiszDane() {
      Student::wypiszDane();
      cout << "Nr dyplomu: " << nrDyplomu << endl;
    }
    int nrDyplomu;
};
```

Klasy `Student` i `Absolwent` mogą być wykorzystane następująco:

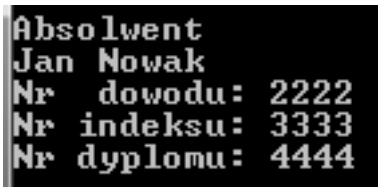
```
Osoba * wo = new Student( "Jan", "Nowak", 2222, 3333 );
. . .
wo->wypiszDane();
. . .
delete wo;
. . .
```

```
Osoba * wo = new Absolwent( "Jan", "Nowak", 2222, 3333, 4444 );
. . .
wo->wypiszDane();
. . .
delete wo;
. . .
```

Zwróćmy uwagę, że mimo wykorzystania wskaźnika do klasy bazowej `Osoba`, każdorazowo wywoła się właściwa wersja funkcji `wypiszDane`, funkcja ta jest bowiem funkcją wirtualną. Wyniki działania wywołań tej funkcji prezentują rys. 24 oraz 25.



Rysunek 24: Wynik działania funkcji *wypiszDane* klasy *Student*



Rysunek 25: Wynik działania funkcji *wypiszDane* klasy *Absolwent*

Funkcje wirtualne w akcji — ewidencja osób

Załóżmy, że chcemy zbudować ewidencję osób, w której będą mogły występować obiekty każdej ze zdefiniowanych wyżej klas. Ewidencję tę realizować będzie klasa `EwidencjaOsob`. Zanim przedstawimy szczegóły jej realizacji, zobaczmy jaka będzie koncepcja je wykorzystania. Rozpocząć musimy od zdefiniowania obiektu klasy `EwidencjaOsob`:

```
EwidencjaOsob ewidencja;
```

Obiekt `ewidencja` będzie zarządzał przechowywanymi obiektami. Funkcja składowa: `void dodaj( Osoba * wo );` będzie dopisywała obiekt do ewidencji. Jej parametrem jest wskaźnik na obiekt klasy `Osoba`. Wiemy już, że taki wskaźnik może przyjmować również wskazania na obiekty dowolnej klasy pochodnej od `Osoba`. Zatem funkcję tę będzie można wywołać w następujący sposób:

```
ewidencja.dodaj( new Osoba( "Andrzej", "Kowalski", 2222 ) );
ewidencja.dodaj( new Student( "Jan", "Nowak", 3333, 4444 ) );
ewidencja.dodaj( new Absolwent( "Anna","Nowak",5555,6666,7777 ) );
```

Spowoduje to dopisanie do ewidencji trzech obiektów, dla każdego z nich pamięć jest przydzielana dynamicznie operatorem `new`[16]. Klasa `EwidencjaOsob` zapamiętuje wskaźnik na przydzielony dynamicznie obiekt. W każdym momencie możemy sprawdzić ile obiektów jest w ewidencji, służy do tego funkcja składowa `podajLbOsob`:

[16] Dla klarowności przykładu nie kontrolujemy poprawności przydziału pamięci.

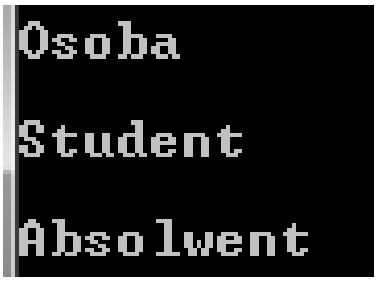
```
cout << "\nLiczba osob: " << ewidencja.podajLbOsob() << endl;
```

Do pobierania informacji o osobach zapisanych w ewidencji służy funkcja składowa: `Osoba * podajOsobe( int nrOsoby )` której rezultatem jest wskaźnik na obiekt o numerze przekazanym parametrem `nrOsoby`. W przypadku podanie numeru spoza zakresu, rezultatem tej funkcji będzie wskaźnik pusty. Pierwsza osoba dodana do ewidencji otrzyma numer 0, druga 1, numer ostatniej osoby w ewidencji wyznaczamy z pomocą przedstawionej wyżej funkcji: `ewidencja.podajLbOsob() - 1`. Pozwoli nam to na programowe przemaszerowanie po elementach ewidencji i poproszenie każdego z obiektów o wyprowadzenie do strumienia wyjściowego informacji o klasie, której jest reprezentantem:

```
for( int i = 0; i < ewidencja.podajLbOsob(); i++ )
  ewidencja.podajOsobe( i )->piszKimJestes();
```

Iteracja `for` rozpoczyna swoje działanie z wartością zmiennej `i` równą zero — jest to numer potencjalnego, pierwszego elementu w ewidencji. Potencjalnego, bowiem gdyby ewidencja była pusta, warunek iteracji byłby fałszywy i iteracja zostałaby zakończona. W naszej ewidencji posiadamy trzy obiekty, zatem iteracja wykona się trzykrotnie.

W ciele iteracji następuje wywołanie funkcji składowej `podajOsobe`. Zmienna `i` zawiera w każdym przebiegu numer kolejnej osoby zapisanej w ewidencji. Rezultatem wywoływanej funkcji jest wskaźnik na ewidencjonowany obiekt, możemy zatem dla rezultatu funkcji wywołać funkcję składową `piszKimJestes()`. Ponieważ jest to funkcja wirtualna, każdy z obiektów wywoła własną wersję tej funkcji, co ilustruje rys. 26.



Rysunek 26: Wynik działania wywołań funkcji *piszKimJestes*

Ciało iteracji `for` można zapisać inaczej — z wykorzystaniem roboczego wskaźnika `wo` zapamiętującego rezultat funkcji. Możemy również sprawdzić, czy ten rezultat nie jest wskaźnikiem pustym:

```
for( int i = 0; i < ewidencja.podajLbOsob(); i++ ) {
  Osoba * wo = ewidencja.podajOsobe( i );
  if( wo )
    wo->piszKimJestes();
}
```

W przypadku właściwej realizacji klasy `EwidencjaOsob` oraz ostrożnego manipulowania zakresem zmiennej sterującej `i` kontrolowanie rezultatu funkcji nie jest konieczne, pierwsza wersja iteracji `for` jest wystarczająca.

W analogiczny sposób możemy zapisać iterację wyprowadzającą do strumienia wyjściowego dane każdego obiektu zapisanego w ewidencji:

```
for( int i = 0; i < ewidencja.podajLbOsob(); i++ )
  ewidencja.podajOsobe( i )->wypiszDane();
```

Ponieważ funkcja `wypiszDane` jest wirtualną, w trakcie każdego przebiegu iteracji zostanie wywołana wersja funkcji właściwa dla obiektu zapisanego w ewidencji, co ilustruje rys. 27.

Rysunek 27: Dane obiektów zapisanych w ewidencji

Podsumujmy — obiekt klasy `EwidencjaOsob` zarządza zbiorem obiektów. Do ewidencji można dodawać obiekty klasy `Osoba` lub dowolnej klasy pochodnej. Ewidencja ta jest zatem *polimorficzna* — zawierać może obiekty występujące w różnych postaciach, wspólnym mianownikiem jest przynależność do wspólnej hierarchii klas. Jeżeli w każdej z klas potomnych klasy `Osoba` dokonana zostanie redefinicja funkcji wirtualnych, mimo owej polimorficzności, każdy z ewidencjonowanych obiektów zachowa zdolność do własnego zachowania — mimo iż operujemy wskaźnikami do klasy `Osoba`, wywołują się funkcje właściwe dla klas każdego z obiektów.

Przyjrzyjmy się teraz bliżej klasie `EwidencjaOsob`. Do przechowywania informacji o ewidencjonowanych obiektach wykorzystana zostanie tablica wskaźników na obiekty klasy `Osoba`. Tablica ta — o nazwie `tabOsob` jest chronionym polem klasy `EwidencjaOsob`. Jej maksymalny rozmiar określa statyczne pole `maksLbOsob`, będące zmienną typu `const int` o wartości 20. [17] Liczbę obiektów aktualnie przechowywanych w ewidencji zawiera pole `lbOsob`. Pola te zostały zadeklarowane w sekcji `protected` klasy, tak by były dostępne dla potencjalnych klas pochodnych. Sekcja chroniona klasy `EwidencjaOsob` ma zatem następującą postać:

[17] Rozwiązanie wykorzystujące tablicę o ustalonych sztywno rozmiarach zastosowano w celu zachowania prostoty przykładu. Elastyczniejszym rozwiązaniem byłoby przykładowo wykorzystanie klasy szablonowej *vector*.

```
class EwidencjaOsob {
   . . .
  protected:
    static const int maksLbOsob = 20;
    Osoba * tabOsob[ maksLbOsob ];
    int lbOsob;
};
```

Klasa powinna posiadać swój konstruktor. W naszym przypadku będzie on jedynie inicjował wartością 0 pole `lbOsob`. Oznacza to, że w tablicy `tabOsob` o maksymalnej liczebności `maksLbOsob` nie zapisano jeszcze żadnego obiektu — ewidencja jest pusta:

```
EwidencjaOsob::EwidencjaOsob() : lbOsob( 0 ) {}
```

Dopisywanie obiektów do ewidencji będzie polegało na wstawianiu wskaźnika na dopisywany obiekt w pierwsze puste miejsce tablicy, pod warunkiem że takie istnieje. Realizuje to funkcja `dodaj` o następującej deklaracji:

```
void EwidencjaOsob::dodaj( Osoba * wo ) {
  if( lbOsob < maksLbOsob - 1 )
    tabOsob[ lbOsob++ ] = wo;
}
```

Jeżeli w tablicy `tabOsob` istnieje wolne miejsce, wartość zmiennej `lbOsob` określa indeks tego wolnego elementu. Dzieje się tak, ponieważ obiekty numerowane są od 0, zatem zmienna ta zawiera dwie istotne informacje — liczbę obiektów aktualnie zapisanych w ewidencji, a jednocześnie numer pierwszego wolnego miejsca w tablicy `tabOsob`. Wskaźnik przekazany parametrem `wo` zostaje w to miejsce zapisany, wartość zmiennej `lbOsob` jest inkrementowana.

Funkcja `podajOsobe` posiada następującą implementację:

```
Osoba * EwidencjaOsob::podajOsobe( int nrOsoby ) {
  if( nrOsoby >= 0 && nrOsoby < lbOsob )
    return tabOsob[ nrOsoby ];
  else
    return 0;
}
```

Jeżeli — tak jak w rozważanym wyżej przypadku — obiekty wstawiane do ewidencji są przydzielane dynamicznie, można je usunąć wywołaniem funkcji `usun`:

```
void EwidencjaOsob::usun() {
  while( --lbOsob >= 0 )
    delete tabOsob[ lbOsob ];
  wyczysc();
}
```

Iteracja `while` fizycznie usuwa obiekty w kolejności od ostatniego do pierwszego. Po usunięciu obiektów zmienna `lbOsob` powinna zostać wyzerowana, ewidencja jest bowiem pusta. Realizuje to funkcja `wyczysc`, która wyłącznie zeruje zmienną przechowującą aktualną liczbę osób:

```
void EwidencjaOsob::wyczysc() {
  lbOsob = 0;
}
```

Wywołanie funkcji `usun` powoduje zatem fizyczne usunięcie zarządzanych obiektów z pamięci operacyjnej. Funkcja `wyczysc` powoduje wyzerowanie liczby przechowywanych obiektów bez ich usuwania z pamięci.

Klasa `EwidencjaOsob` może wykonywać operacje na przechowywanych obiektach, przykładowo funkcja `wypiszWszystko` wyprowadza do strumienia wyjściowego informację o każdym obiekcie zapisanym w ewidencji:

```
void EwidencjaOsob::wypiszWszystko() {
  for( int i = 0; i < lbOsob ; i++ )
    tabOsob[ i ]->wypiszDane();
}
```

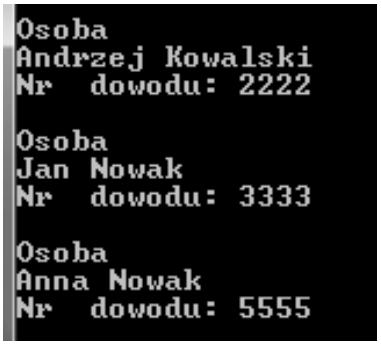
Zwróćmy ponownie uwagę, że informacje wyprowadzane do strumienia wyjściowego programu zależą od klasy obiektu wskazywanego przez element tablicy `tabOsob[ i ]`. Dzieje się tak, ponieważ funkcja `wypiszDane` jest wirtualna. Dzięki temu rezultat działania funkcji `wypiszWszystko` wywołanej dla obiektu `ewidencja` w rozważanym wcześniej przykładzie będzie taki jak na rys. 27. Przypomnijmy:

```
EwidencjaOsob ewidencja;
. . .
ewidencja.dodaj( new Osoba( "Andrzej", "Kowalski", 2222 ) );
ewidencja.dodaj( new Student( "Jan", "Nowak", 3333, 4444 ) );
ewidencja.dodaj( new Absolwent( "Anna","Nowak",5555,6666,7777 ) );
. . .
ewidencja.wypiszWszystko();
```

Zauważmy, że usunięcie słowa `virtual` z definicji obu funkcji składowych klasy `Osoba` zmienia wszystko — funkcje są wiązane

statycznie a wynik działania powyższej funkcji byłby taki jak na rys. 28.

Bez funkcji wirtualnych znika polimorfizm — wszystkie ewidencjonowane obiekty zachowują się tak, jakby były obiektami klasy `Osoba`. Polimorfizm nie jest zatem cechą klasy `EwidencjaOsob` a właściwością hierarchii klas, zbudowanej na klasie `Osoba`. Stosowanie polimorfizmu wymaga przewidywania — w trakcie budowania klas należy przewidzieć czy będą one wykorzystywane jako klas bazowe, oraz jakie funkcje składowe powinny być wirtualnymi.



Rysunek 28: Działanie funkcji *wypiszWszystko* dla wiązania statycznego

Poniżej przedstawiona została kompletna deklaracja klasy `EwidencjaOsób` oraz definicje dwóch, nieomówionych wcześniej funkcji składowych.

```
class EwidencjaOsob {
  public:
    EwidencjaOsob();
    void dodaj( Osoba * wo );
    Osoba * podajOsobe( int nrOsoby );
    void wypisz( int nrOsoby );
    void wypiszWszystko();
    int podajLbOsob();
    void usun();
    void wyczysc();

  protected:
    static const int maksLbOsob = 20;
    Osoba * tabOsob[ maksLbOsob ];
    int lbOsob;
};

void EwidencjaOsob::wypisz( int nrOsoby ) {
  if( nrOsoby >= 0 && nrOsoby < lbOsob )
    tabOsob[ nrOsoby ]->wypiszDane();
}

int EwidencjaOsob::podajLbOsob() {
  return lbOsob;
}
```

Konwersje wskaźników do obiektów

Obiektami zapisanymi w ewidencji osób można manipulować. Załóżmy, że dla ewidencji osób przedstawionej na str. 56 chcemy zmienić imię studenta Nowaka z Jan na Janusz. Dostęp do drugiego obiektu w ewidencji uzyskamy za pośrednictwem wywołania funkcji `podajOsobe` z numerem 1.

```
Osoba * wo = ewidencja.podajOsobe( 1 );
if( wo )
  wo->imie = "Janusz";
```

W podobny sposób zmienimy nazwisko i numer dowodu każdej z osób zapisanych w ewidencji. Jeżeli spróbujemy jednak zmienić w analogiczny sposób numer indeksu: `wo->nrIndeksu = 1234;` kompilator zgłosi błąd kompilacji, mówiący o tym, że klasa `Osoba` nie posiada pola `nrIndeksu`. To prawda, posługujemy się bowiem wskaźnikiem do klasy `Osoba` — taki jest rezultat funkcji `podajOsobe`. Spróbujmy zatem użyć wskaźnika do klas `Student`:

```
Student * ws = ewidencja.podajOsobe( 1 );
if( ws )
  ws->nrIndeksu = 1234
```

Niestety, mimo iż rzeczywiście drugi obiekt w ewidencji jest obiektem klasy `Student`, kompilator nie pozwoli na bezpośrednie przypisanie wskaźnika klasy `Osoba` do wskaźnika klasy `Student` — zostanie zgłoszony błąd kompilacji. Aby takie przypisanie mogło być zrealizowane, programista musi dokonać jawnej konwersji wskaźnika będącego rezultatem funkcji `podajOsobe`:

```
Student * ws = ( Student * )ewidencja.podajOsobe( 1 );
if( ws )
  ws->nrIndeksu = 1234
```

W pierwszej linii powyższego przykładu następuje przekształcenie typu wskaźnika będącego rezultatem wywołania funkcji `podajOsobe`. Zapis `( Student * )` oznacza, że rezultat funkcji ma zostać przekształcony do postaci wskaźnika do klasy `Student`. Po tak przeprowadzonej konwersji typu wskaźnika, może on zostać przypisany do zmiennej wskźnikowej `ws`. Tak realizowana konwersja, zwana również *rzutowaniem*, pochodzi wprost z języka C.

Dokonana konwersji wskaźnika pozwala na późniejsze odwoływanie się do składowych obiektu klasy `Student`. Konwersja taka jest jednak bezpieczna tylko wtedy, gdy rzeczywiście wskazywany obiekt należy do klasy `Student`. Załóżmy, że w powyższym przykładzie wywołujemy funkcję `podajOsobe` z parametrem 0. Pierwszy element w ewidencji jest obiektem klasy `Osoba`, po dokonaniu konwersji na wskaźnik do klasy `Student` otrzymamy możliwość odwołania się do pola `nrIndeksu`, którego w obiekcie klasy `Osoba` nie ma! Takie odwołania do nieistniejących elementów klasy są zawsze błędne, ich efekty są trudno przewidywalne i mocno zależne od środowiska systemowego. W większości przypadków spowodują one awaryjne zakończenie programu, chociaż nie jest wykluczone, że program nie zostanie zakończony i będzie działał dalej, zwykle błędnie.

Aby uniknąć tego typu błędów, należy opracować mechanizm identyfikacji klas obiektów przechowywanych w ewidencji, pozwalający na stwierdzenie, czy mamy do czynienia z obiektem klasy `Osoba`, `Student` czy `Absolwent`. Taka identyfikacja może zostać zrobiona na dwa sposoby:

- przez wbudowanie przez programistę w poszczególne klasy informacji o ich rodzaju,

- przez wykorzystanie mechanizmów języka C++ pozwalających na identyfikowanie typów w czasie wykonania programu.

Każdy z tych sposobów posiada swoje wady i zalety, oba rozwiązania przedstawione są w kolejnych podrozdziałach.

Identyfikacja typów zdefiniowana przez programistę

To rozwiązanie zakłada, że programista wyposaża tworzone przez siebie klasy w elementy pozwalające na identyfikację klasy obiektu w trakcie wykonania programu. Nie istnieje jedynie słuszne rozwiązanie tego problemu, istnieje kilka możliwości. Przedstawione dalej rozwiązanie wykorzystuje funkcje wirtualne. Jego podstawowym elementem będzie typ wyliczeniowy, którego wartości będą odpowiadały nazwom poszczególnych klas wchodzących w skład hierarchii dziedziczenia:

```
enum RodzajeOsob {
  roOsoba,
  roStudent,
  roAbsolwent
}
```

Prefiks `ro` to akronim sformułowania *rodzaj osoby*. Każdą z klas w hierarchii uzupełnimy o funkcję `rodzaj`, której rezultatem będzie jedna z wartości powyższego typu wyliczeniowego:

```
class Osoba {
  public:
    . . .
    virtual int rodzaj() { return roOsoba; }
    . . .
};

class Student : public Osoba {
  public:
    . . .
    int rodzaj() { return roStudent; }
    . . .
};

class Absolwent  : public Student {
  public:
    . . .
    int rodzaj() { return roAbsolwent; }
    . . .
};
```

Przypomnijmy, że rozważania przedstawione w tym rozdziale dotyczą ewidencji o strukturze utworzonej w przykładzie przedstawionym na str. 56. Zauważmy, że funkcja `rodzaj` jest wirtualna.

Tak zdefiniowane funkcje pozwolą na identyfikację klasy obiektu w trakcie wykonania programu:

```
for( int i = 0; i < ewidencja.podajLbOsob(); i++ ) {
  Osoba * wo = ewidencja.podajOsobe( i );
  if( wo->rodzaj() == roOsoba )
    cout << "\nObiekt klasy Osoba";
  if( wo->rodzaj() == roStudent )
    cout << "\nObiekt klasy Student";
  if( wo->rodzaj() == roAbsolwent )
    cout << "\nObiekt klasy Absolwent";
}
```

Identyfikacja klasy obiektu pozwala nam na bezpieczne odwoływanie się do składowych jemu właściwych:

```
Osoba * wo = ewidencja.podajOsobe( 1 );
if( wo->rodzaj() == roStudent ) {
  Student * ws = ( Student * ) wo;
  ws->nrIndeksu = 1234;
}
```

Powyższy fragment można zapisać bez wykorzystania pomocniczego wskaźnika do klasy `Student`, bezpośrednio odwołując się do wskaźnika `wo` po jego konwersji:

```
Osoba * wo = ewidencja.podajOsobe( 1 );
if( wo->rodzaj() == roStudent )
  ( ( Student * ) wo )->nrIndeksu = 1234;
```

Zaprezentowane rozwiązanie identyfikacji klasy obiektu w trakcie wykonania programu bazuje na funkcjach wirtualnych. Jest to rozwiązanie proste, skuteczne i efektywne czasowo[18]. Wymaga ono jednak przewidywania — już na etapie konstruowania klasy bazowej trzeba przewidzieć taką możliwość oraz konsekwentnie redefiniować odpowiednią funkcję wirtualną w kasach pochodnych.

[18] Z uwzględnieniem oczywiście narzutu czasowego związanego z wywołaniem funkcji wirtualnej.

Identyfikacja typów jako element języka

Problem identyfikowania typów obiektów w czasie wykonania programu dostrzeżono w trakcie prac nad rozwojem języka C++, rozszerzając go o taki właśnie mechanizm.

Konwersja typów może być realizowana poprzez wykorzystanie operatorów konwersji. Pierwszy z nich zastępuje klasyczną konwersję wywodzącą się z języka C, stosowaną w poprzednim podrozdziale. Zamiast konwersji w postaci:

```
Student ws = ( Student * )ewidencja.podajOsobe(1);
```

piszemy

```
Student ws = static_cast< Student * >(ewidencja.podajOsobe(1));
```

Operator `static_cast` działa podobnie jak rzutowanie pochodzące z języka C[19]. Operator ten nie realizuje żadnej kontroli typów w trakcie działania programu i jego stosowanie stwarza podobne niebezpieczeństwa jak zwykła konwersja typów.

[19] W istocie najlepszym odpowiednikiem klasycznego rzutowania wywodzącego się z języka C jest operator *reinterpret_cast*, który dokonuje określonych przez programistę konwersji, również ryzykownych i potencjalnie niebezpiecznych.

Z tego powodu wprowadzono w języku C++ operator `dynamic_cast`. Operator ten pozwala na konwersję wskaźników i referencji na obiekty polimorficzne z kontrolą, czy taka konwersja może być przeprowadzona. W przedstawionym niżej przykładzie rezultat funkcji `podajOsobe` — a więc wskaźnik na obiekt klasy `Osoba` — podawany jest na wejście operatora `dynamic_cast`. Jeżeli możliwa jest konwersja do postaci wskaźnika na klasę pochodną `Student`, operator taką konwersję realizuje. Jeżeli konwersja nie może zostać zrealizowana, rezultatem operatora jest wartość zerowa. Konwersja taka jest możliwa, jeżeli wskaźnik aktualnie wskazuje na obiekt klasy `Student` lub obiekt jej klasy pochodnej.

```
Student * ws = dynamic_cast< Student * >(ewidencja.podajOsobe(1));
if( ws != 0 )
  ws->nrIndeksu = 1234;
```

W powyższym przykładzie, drugi element w ewidencji jest rzeczywiście obiektem klasy `Student`, zatem konwersja jest możliwa. Gdyby funkcja `podajOsobe` została wywołana z parametrem 0, rezultatem operatora byłaby wartość zerowa — pierwszy element w ewidencji jest obiektem klasy `Osoba` i konwersja wskaźnika nie jest możliwa. Zwróćmy uwagę, że konwersja powiedzie się również w przypadku wywołania funkcji `podajOsobe` z parametrem 2. Trzeci element ewidencji jest bowiem obiektem klasy `Absolwent`, a ta jest pochodną klasy `Student`. Trzeci obiekt w ewidencji posiada więc wszystkie właściwości klasy `Student`, a zatem i pole `nrIndeksu`.

Operator `dynamic_cast` kontroluje możliwość wykonania konwersji w trakcie działania programu, opierając się na informacjach o typach obiektów dołączonych do skompilowanego programu. Mechanizm ten został wprowadzony do języka C++ w pierwszej połowie lat dziewięćdziesiątych ubiegłego wieku, znany jest on jako Run-Time Type Information, w skrócie RTTI. Mimo iż ten mechanizm nie jest nowy, ciągle jego realizacja może się różnić w zależności od kompilatora. W niektórych kompilatorach mechanizm ten jest domyślnie wyłączony (np. Visual C++), aby skorzystać z jego możliwości należy ustawić odpowiednią opcję kompilacji. Dokładne omówienie RTTI przekracza ramy niniejszego opracowania — zapamiętajmy, że z punktu widzenia sprawnego stosowania metod wirtualnych wystarczy rozważne stosowanie operatora `dynamic_cast`.

Własna czy wbudowana kontrola typów?

Kontrolowanie rzeczywistego typu obiektu polimorficznego może być realizowane poprzez wbudowanie przez programistę dodatkowych informacji do każdej z klas w hierarchii — tak jak w przypadku omówionej wcześniej funkcji `rodzaj`. Rozwiązanie alternatywne

wykorzystuje RTTI i operator `dynamic_cast`. Które rozwiązanie wybrać? Odpowiedź nie jest jednoznaczna. Mechanizm RTTI jest wygodny wtedy, gdy pracujemy z klasami których nie implementowaliśmy sami. Gdy nie zostały wyposażone w środki identyfikacji konkretnego wcielenia obiektu polimorficznego, RTTI pozwala na taką identyfikację. Przemyślane stosowanie operatora dynamicznej konwersji jest bezpieczne i skuteczne. Rozwiązanie to wymaga jednak działania dynamicznej identyfikacji typów, a to może trwać w przypadku rozbudowanych hierarchii klas.

W przypadku własnej realizacji hierarchii klas warto skorzystać z rozwiązania pierwszego — wprowadzenie własnej identyfikacji, np. opartej na funkcjach wirtualnych, nie jest czasochłonne, nie powiększa znacząco rozmiaru generowanego kodu oraz nie niesie za sobą istotnych narzutów czasowych. Uniezależnienie kodu od RTTI poprawia jego przenośność i pozwala na zastosowanie kompilatora nie posiadającego obsługi RTTI. Dodatkowo własna identyfikacja nie wyklucza wykorzystania RTTI, zwiększając elastyczność tworzonych klas.

Klasy abstrakcyjne

Dziedziczenie oraz polimorfizm to esencjonalne cechy programowania obiektowego. Pozwalają one między innymi na wygodne zarządzanie niejednorodnymi kolekcjami obiektów — co przedstawiono w poprzednim podrozdziale. Zauważmy, jak istotną rolę pełni klasa `Osoba`. Stanowi ona swoisty szablon, określający metodę komunikacji z obiektami tej klasy, jak i klas pochodnych. Taka klasa stanowi *interfejs*, określający metodę komunikacji z obiektami tworzącymi hierarchię klas.

Taką klasę bazową można w języku C++ zbudować w sposób specyficzny — może ona być *klasą abstrakcyjną*. Pojęcie to wprowadzimy, posługując się następującym przykładem.

Załóżmy, że naszym zadaniem jest napisanie oprogramowania monitorującego stan czujników systemu alarmowego, chroniącego pewien budynek. Czujniki mogą być trojakiego rodzaju:

- czujniki otwarcia okien,
- czujniki otwarcia drzwi,
- czujniki ruchu.

Czujników systemu alarmowego jest zwykle wiele, ich liczba i rodzaj zależy od specyfiki budynku (liczby okien, drzwi itp.). Każdy z czujników jest inaczej zbudowany, w naszym programie dla reprezentowania czujnika każdego rodzaju przewidziana będzie osobna klasa. Jednak niezależnie od różnic, z punktu widzenia systemu alarmowego, wszystkie czujniki posiadają podobne cechy:

- w momencie aktywacji alarmu należy je włączyć, będzie to realizowała funkcja składowa `wlacz` wbudowana w każdą klasę opisu czujnika;

- w momencie dezaktywacji alarmu należy je wyłączyć, będzie to realizowała funkcja składowa `wylacz` wbudowana w każdą klasę opisu czujnika;
- identyfikacja czujnika będzie realizowana poprzez funkcję `rodzaj`, której rezultatem będzie wartość całkowita identyfikująca rodzaj czujnika;
- do monitorowania stanu czujnika służyć będzie funkcja składowa `stanAlarmowy`, której rezultatem jest wartość `true` gdy czujnik zadziałał (wykryto potencjalne włamanie), zaś wartość `false` w przypadku przeciwnym.

Te wspólne właściwości wszystkich czujników staną się podstawą dla zdefiniowania klasy `Czujnik`. Klasa ta będzie reprezentowała pewien abstrakcyjny czujnik. Na tym etapie nie jest istotna zasada działania, ważne są jednak operacje, jakie będzie można na takim czujniku wykonać. Klasa opisu abstrakcyjnego czujnika może posiadać następującą postać:

```
class Czujnik {
  public:
    Czujnik() {}
    virtual void wlacz() {};
    virtual void wylacz() {};
    virtual int  rodzaj() { return 0 };
    virtual bool stanAlarmowy() { return false };
};
```

Zwróćmy uwagę, że w klasie tej nie zdefiniowano żadnego pola. Posiada ona konstruktor, oraz cztery opisane wcześniej funkcje składowe. Funkcje te są oczywiście funkcjami wirtualnymi. Ponieważ klasa `Czujnik` reprezentuje czujnik abstrakcyjny, trudno wskazać, jakie powinny być ciała owych funkcji. Właściwa implementacja funkcji powinna zostać umieszczona w klasach pochodnych. Dokładniej — każda klasa pochodna *powinna* dokonać redefinicji funkcji składowej, wprowadzając jej implementację adekwatną do określonego typu czujnika.

Klasę `Czujnik` można zdefiniować w specyficzny sposób — jako *klasę abstrakcyjną*. Pozwoli to na uzyskanie dwóch efektów — podkreślenie abstrakcyjnego charakteru klasy `Czujnik` oraz wymuszenie konieczności redefinicji funkcji składowych w klasach pochodnych. W języku C++ buduje się klasę abstrakcyjną poprzez umieszczenie w jej definicji przynajmniej jednej *funkcji abstrakcyjnej*. W oryginale funkcja abstrakcyjna nosi nazwę *pure virtual function*, jednak dosłowne tłumaczenie[20] nie wydaje się szczęśliwe, proponuje się zatem stosowanie nazwy *funkcja abstrakcyjna*. Funkcja abstrakcyjna nie posiada ciała, w jego miejscu występuje symbol „`= 0`”:

[20] Terminy *czysta funkcja wirtualna* czy *w pełni wirtualna funkcja* nie oddają zdaniem autora istoty sprawy i są niewygodne w stosowaniu.

```
class Czujnik {
  public:
```

```
    Czujnik() {}
    virtual void wlacz() = 0;
    virtual void wylacz() = 0;
    virtual int  rodzaj() = 0;
    virtual bool stanAlarmowy() = 0;
};
```

Tak zdefiniowana klasa jest abstrakcyjna, posiada cztery funkcje abstrakcyjne, których właściwa definicja musi zostać umieszczona w klasach pochodnych. Nie wszystkie funkcje muszą być abstrakcyjne — wystarczy jedna, by klasa stała się abstrakcyjną.

Rola klasy abstrakcyjnej jest specyficzna — nie deklaruje się jawnie obiektów tej klasy, służy ona jako klasa bazowa, stając się korzeniem, często bardzo rozbudowanej hierarchii klas. W klasach pochodnych należy umieścić konkretne definicje funkcji abstrakcyjnych. W rozważanym przykładzie, klasa `Czujnik` będzie stanowiła klasę bazową dla klas opisujących czujnik otwarcia okien, drzwi i czujnik ruchu. W klasach tych należy umieścić konkretne realizacje funkcji abstrakcyjnych, adekwatne do roli przypisanej danej klasie. W rozważanym przykładzie rozróżniamy trzy czujniki systemu alarmowego, każda z nich identyfikowana będzie stałą wyliczeniową następującego typu:

```
enum RodzajeCzujnikow {
  rcCzujnikRuchu,
  rcCzujnikOkna,
  rcCzujnikDrzwi
};
```

Klasa opisu czujnika ruchu może mieć poglądową postać umieszczoną niżej. Oczywiście, dla klarowności prezentowanego przykładu, nie jest możliwe przytoczenie w pełni działającego kodu — dlatego np. ciało funkcji `stanAlarmowy` zawiera jedynie słowny komentarz zadania, jakie ta funkcja ma wykonać.

```
class CzujnikRuchu : public Czujnik {
  public:
    CzujnikRuchu() : Czujnik() {}
    void wlacz() { cout << "\nCzujnk ruchu wlaczony"; };
    void wylacz() { cout << "\nCzujnk ruchu wylaczony"; };
    int  rodzaj() { return rcCzujnikRuchu; };
    bool stanAlarmowy() {
      Tu kod testowania stanu czujnika, rezultat true gdy wykryto
      potencjalne włamanie, false w przeciwnym przypadku.
    }
};
```

W podobny sposób zdefiniowano klasy opisu czujnika otwarcia okien i drzwi:

```
class CzujnikOkna : public Czujnik {
```

```
  public:
    CzujnikOkna() : Czujnik() {}
    void wlacz() { cout << "\nCzujnk okna wlaczony"; };
    void wylacz() { cout << "\nCzujnk okna wylaczony"; };
    int  rodzaj() { return rcCzujnikOkna; };
    bool stanAlarmowy() {
      Tu kod testowania stanu czujnika, rezultat true gdy wykryto
      potencjalne włamanie, false w przeciwnym przypadku.
    }
};

class CzujnikDrzwi : public Czujnik {
  public:
    CzujnikDrzwi() : Czujnik() {}
    void wlacz() { cout << "\nCzujnk drzwi wlaczony"; };
    void wylacz() { cout << "\nCzujnk drzwi wylaczony"; };
    int  rodzaj() { return rcCzujnikDrzwi; };
    bool stanAlarmowy() {
      Tu kod testowania stanu czujnika, rezultat true gdy wykryto
      potencjalne włamanie, false w przeciwnym przypadku.
    }
};
```

Załóżmy, że nasze czujniki mają być zainstalowane w budynku z jednymi drzwiami, trzema oknami, wewnątrz mają być zainstalowane trzy czujniki ruchu. Każdy z fizycznych czujników będzie monitorowany przez obiekt odpowiedniej, wyżej zdefiniowanej klasy. Potrzebujemy łącznie siedem czujników, do zarządzania nimi wykorzystamy tablicę wskaźników na obiekty klasy `Czyjnik` (koncepcja jest analogiczna do tej, przedstawionej przy okazji opisu klasy `EwidencjaOsob`):

```
const int maksLbCzujnikow = 20;
Czujnik * tabCzujnikow[ maksLbCzujnikow ];
int lbCzujnikow = 7;
```

Przedstawiony niżej fragment kodu ilustruje utworzenie siedmiu obiektów odpowiednich klas, wskaźniki do utworzonych obiektów zapisane zostaną w tablicy:

```
tabCzujnikow[ 0 ] = new CzujnikRuchu();
tabCzujnikow[ 1 ] = new CzujnikRuchu();
tabCzujnikow[ 2 ] = new CzujnikRuchu();
tabCzujnikow[ 3 ] = new CzujnikDrzwi();
tabCzujnikow[ 4 ] = new CzujnikOkna();
tabCzujnikow[ 5 ] = new CzujnikOkna();
tabCzujnikow[ 6 ] = new CzujnikOkna();
```

Operacja włączenia alarmu — aktywacji wszystkich czujników — może być zrealizowana następująco:

```
for( int i = 0; i < lbCzujnikow ; i++ )
  tabCzujnikow[ i ]->wlacz();
```

Komplementarna zaś operacja wyłączenia alarmu może mieć następującą postać:

```
for( int i = 0; i < lbCzujnikow ; i++ )
  tabCzujnikow[ i ]->wylacz();
```

Usunięcie obiektów reprezentujących czujniki może być zrealizowane następująco:

```
while( --lbCzujnikow >= 0 )
  delete tabCzujnikow[ lbCzujnikow ];
```

Zwróćmy uwagę, że w powyższych fragmentach kodu wykorzystujemy funkcje zdefiniowane w abstrakcyjnej klasie `Czujnik`. Odwołania wykonywane są za pośrednictwem wskaźników do dynamicznie utworzonych obiektów. Są to funkcje wirtualne, mamy zatem pewność, że wywołane zostaną wersje właściwe dla konkretnego obiektu. Dzięki temu, że są to funkcje abstrakcyjne, mamy również pewność, że rzeczywiście zdefiniowano je w klasach pochodnych.

Operację monitorowania stanu czujników można zrealizować następująco:

```
while( systemAktywny() ) {
  for( int i = 0; i < lbCzujnikow; i++ )
    if( tabCzujnikow[ i ]->stanAlarmowy() )
      zglosAlarm( tabCzujnikow[ i ] );
}
```

Iteracja `while` wykonuje się, dopóki nie zostanie wykryta dezaktywacja systemu alarmowego. Gdy system jest aktywny, rezultatem funkcji `systemAktywny` jest wartość `true`, wyłączenie systemu spowoduje, że rezultatem tej funkcji będzie wartość `false`. W trakcie każdego przebiegu iteracji `while` przeglądana jest tablica obiektów reprezentujących czujniki. Jeżeli któryś z obiektów wykrył zadziałanie czujnika, rezultatem jego funkcji składowej `stanAlarmowy` będzie wartość `true`. Spowoduje to wywołanie funkcji `zglosAlarm`. Funkcja ta otrzymuje jako parametr wskaźnik na obiekt, reprezentujący czujnik, który zgłosił sytuację alarmową. Funkcja ta na podstawie rezultatu wywołania funkcji `rodzaj` określa z jakim typem czujnika mamy do czynienia i wyprowadza do strumienia wyjściowego odpowiedni komunikat:

```
void zglosAlarm( Czujnik * wc ) {
  cout << "\nWykrytu zgloszenie z czujnika ";
  switch( wc->rodzaj() ) {
    case rcCzujnikRuchu : cout << "ruchu!" << endl;
                          break;
    case rcCzujnikOkna  : cout << "otwarcia okna!" << endl;
```

```
                        break;
    case rcCzujnikDrzwi : cout << "otwarcia drzwi!" << endl;
                        break;
  }
}
```

Zauważmy, że odwołując się do czujników cały czas bazujemy na funkcjach zdefiniowanych w klasie abstrakcyjnej `Czujnik`. Określają one sposób wymiany informacji z wszystkimi specjalizowanymi czujnikami, reprezentowanymi przez obiekty klas pochodnych. Klasa `Czujnik` stanowi zatem *interfejs*, definiujący sposób obsługi i wymiany informacji z czujnikami.

Zaprezentowane wcześniej rozwiązanie zarządzania wieloma czujnikami jest skuteczne lecz nieeleganckie. Do obsługi czujników możemy zaprojektować specjalną klasę — niech nazywa się ona `SystemAlarmowy`. Zdefiniujmy obiekt `system`, tej właśnie klasy.

```
SystemAlarmowy system;
```

Dopisanie do systemu obiektów reprezentujących czujniki realizowane jest za pośrednictwem funkcji `dodajCzujnik`:

```
system.dodajCzujnik( new CzujnikRuchu() );
system.dodajCzujnik( new CzujnikRuchu() );
system.dodajCzujnik( new CzujnikRuchu() );
system.dodajCzujnik( new CzujnikDrzwi() );
system.dodajCzujnik( new CzujnikOkna() );
system.dodajCzujnik( new CzujnikOkna() );
system.dodajCzujnik( new CzujnikOkna() );
```

Włączenie czujników, przejście w stan monitorowania oraz wyłączenie czujników po dezaktywacji systemu może mieć następującą postać:

```
system.wlaczCzujniki();
system.monitoruj();
system.wylaczCzujniki();
```

Usunięcie utworzonych obiektów reprezentujących czujniki realizuje wywołanie:

```
system.usunCzujniki();
```

Klasa `SystemAlarmowy` przechowuje wskaźniki do obiektów reprezentujących czujniki w tablicy wskaźników do klasy `Czujnik`. Wszystkie operacje na zapisanych tam czujnikach są wykonywane za pośrednictwem interfejsu zdefiniowanego poprzez funkcje abstrakcyjne tejże klasy. Definicja klasy ma następująca postać:

```
class SystemAlarmowy {
  public:
    SystemAlarmowy();
    void dodajCzujnik( Czujnik * wc );
```

```
    Czujnik * podajCzujnik( int nr );
    void wlaczCzujniki();
    void monitoruj();
    void wylaczCzujniki();
    void zglosAlarm( Czujnik * wc );
    bool systemAktywny();
    void usunCzujniki();
    void wyczysc();

  protected:
    static const int maksLbCzujnikow = 20;
    Czujnik * tabCzujnikow[ maksLbCzujnikow ];
    int lbCzujnikow;
};
```

Definicja konstruktora, funkcji dodawania wskaźnika oraz pobierania informacji o nim, mają następującą postać:

```
SystemAlarmowy::SystemAlarmowy() : lbCzujnikow( 0 ) {}

void SystemAlarmowy::dodajCzujnik( Czujnik * wc ) {
  if( lbCzujnikow < maksLbCzujnikow - 1 )
    tabCzujnikow[ lbCzujnikow++ ] = wc;
}

Czujnik * SystemAlarmowy::podajCzujnik( int nrCzujnika ) {
  if( nrCzujnika >= 0 && nrCzujnika < lbCzujnikow )
    return tabCzujnikow[ nrCzujnika ];
  else
    return 0;
 }
```

Operację włączenia i wyłączenia czujników wykonywane są w sposób opisany wcześniej, wykonują je następujące funkcje:

```
void SystemAlarmowy::wlaczCzujniki() {
  for( int i = 0; i < lbCzujnikow ; i++ )
    tabCzujnikow[ i ]->wlacz();
}

void SystemAlarmowy::wylaczCzujniki() {
  for( int i = 0; i < lbCzujnikow; i++ )
    tabCzujnikow[ i ]->wylacz();
}
```

Analogicznie jest również zrealizowana funkcja monitorowania stanu czujników:

```
void SystemAlarmowy::monitoruj() {
  while( systemAktywny() ) {
    for( int i = 0; i < lbCzujnikow; i++ )
```

```
      if( tabCzujnikow[ i ]->stanAlarmowy() )
        zglosAlarm( tabCzujnikow[ i ] );
  }
}
```

Funkcje `systemAktywny` oraz `zglosAlarm` realizują operacje opisane wcześniej.

Klasa `SystemAlarmowy` jest przygotowana na obsługę dowolnych urządzeń, dziedziczących interfejs określony klasą `Czujnik`. Załóżmy, że chcemy rozbudować system alarmowy o czujniki pożarowe, wykrywające dym. Nic nie stoi na przeszkodzie zdefiniowania kolejnej klasy pochodnej — np. `CzujnikDymu` — która po implementacji funkcji abstrakcyjnych może być wykorzystana do tworzenia kolejnych obiektów, dodawanych do listy monitorowanych czujników.

Podsumowanie

Język C++ ciągle rozwija się i ewoluuje. Aktualnie obowiązujący standard języka C++ oznaczany jest symbolem C++0x[21]. Przedstawione w poprzednich rozdziałach mechanizmy programowania obiektowego dostępne są już od wczesnych wersji języka C++, stanowią podstawę obiektowego podejścia do programowania w tym języku. Większość tych mechanizmów została przeniesiona do innych języków programowania — np. do języka Java, C# czy PHP. Dobre zrozumienie opisanych wcześniej rozwiązań obiektowych ułatwi naukę kolejnych języków programowania obiektowego, oraz wychwycenie — czasem bardzo subtelnych — różnic w pojmowaniu i realizacji obiektowości w każdym z nowoczesnych języków obiektowych.

[21] Znany również jako C++11.

Przedstawione rozwiązania w zakresie obiektowości dalekie są od wyczerpania tematu — język C++ zawiera jeszcze wiele istotnych mechanizmów obiektowych. Niestety, ograniczone ramy niniejszego opracowania nie pozwalają na szczegółowe ich omówienie. Istotne znaczenie mają szczególnie dwa mechanizmy — przeciążanie operatorów (ang. operator overloading) oraz stosowanie szablonów (ang. templates). Stanowią one interesującą podstawę dla konstrukcji klas kontenerowych, oferujących najróżniejsze możliwości organizowania struktur danych. Aktualnie standardem stało się wykorzystanie biblioteki STL (ang. Standard Template Library), która oferuje szereg możliwości przechowywania oraz manipulowania danymi.

Należy zwrócić uwagę, że koncepcja obiektowości obecna w języku C++ nie jest jedyną i wzorcową. Istnieje wiele różniących się w szczegółach realizacji koncepcji obiektowości — np. w języku Smalltalk, Objective-C czy Ruby. Jednak podejście oferowane przez język C++ jest sprawdzone praktycznie i potwierdzone praktyką realizacji ogromnej liczby projektów programistycznych — o różnej skali i charakterze. Koncepcja ta wspierana jest przez producentów kompilatorów i środowisk programistycznych, również tych, które umożliwiają szybkie prototypowanie programów — środowisk RAD

(ang. Rapid Application Development). Środowiska takie, mające wcześniej charakter rozwiązań komercyjnych (np. Microsoft Visual C++, Borland C++ Builder) zaczynają być dostępne w wersjach dostarczanych darmowo, bazujących na rozwiązaniach typu oprogramowania otwartego (np. QtCreator firmowany aktualnie przez firmę Diga).

Inwestycja w nabycie umiejętności sprawnego programowania obiektowego w języku C++ wydaje się inwestycją trafioną. Jednak należy zwrócić uwagę na to, że język C++ jest rzeczywiście językiem skomplikowanym, a aktualne tendencje w programowaniu w C++ oraz aktualny standard języka zdają się to potwierdzać. Codziennością staje się wykorzystanie szablonów, klas z biblioteki STL — ten kierunek wydaje się znacznie upraszczać kod i nakład pracy programisty, jednak wymaga od niego drobiazgowej znajomości nie tylko samego języka, ale również właśnie biblioteki STL[22].

[22] Poza biblioteką STL, która jest niekwestionowanym standardem w dziedzinie podstawowych struktur danych i algorytmów, warto również zapoznać się z biblioteką Boost (`www.boost.org`), zawierającą wiele użytecznych pakietów m.in. do obsługi wyrażeń regularnych, odczytu i zapisu dokumentów XML czy komunikacji sieciowej.

Istnieje drugi czynnik sprawiający, że droga od podstaw C++ do sprawnego programowania aplikacji komercyjnych w tym języku jest długa i żmudna. Sam język jak i jego standardowe biblioteki nie wspierają operacji programowania GUI, dostępu do baz danych, programowania sieciowego. Operacje te realizują specjalizowane biblioteki, dostarczane zwykle z określonym środowiskiem programistycznym (np. VCL i C++ Builder). Rozwiązania te są zwykle nieprzenośne na poziomie kodu źródłowego. Powstają jednak niekomercyjne rozwiązania typu biblioteka WxWidgets czy Qt. Biblioteki te oferują pełny zakres narzędzi (najczęściej w postaci bibliotek klas) brakujących językowi C++, a koniecznych do tworzenia w pełni wartościowego oprogramowania.

Wskazane powyżej czynniki sprawiają, iż coraz popularniejsze stają się języki pochodne od C++, takie jak Java i C#, w których zrezygnowano z wielu złożonych konstrukcji i które rdzennie zapewniają wsparcie dla programowania GUI czy dostęp do baz danych. Uzupełnione sprawnymi środowiskami RAD, języki te stają się interesująca alternatywą dla C++[23]. Jednak — zdaniem autora — nie należy się obawiać o przyszłość tego języka i spadek jego popularności. Oferowana przez C++ elastyczność, szybkość wsparta nowoczesnymi bibliotekami typu Qt sprawiają, że język ten jeszcze na długi czas pozostanie podstawowym narzędziem programowania dla profesjonalistów.

[23] Rozważamy tutaj aspekt tworzenia oprogramowania klasy desktop — liczba rozwiązań alternatywnych w segmencie narzędzi do tworzenia aplikacji internetowych jest wyraźnie większa.

Zadania

Zad. 1. Zdefiniować klasę `Data` zawierającą pola prywatne: `int dzien`, `int miesiac`, `int rok`. Klasa powinna zawierać funkcje dostępowe, pobierające zawartość poszczególnych pól (np. `podajDzien` itd.), oraz funkcje pozwalające na ustalanie zawartości pól (np. `ustawDzien`, itd.). Klasa ma posiadać konstruktor domyślny, zerujący wszystkie pola, oraz konstruktor ogólny, otrzymujący trzy parametry, trafiające odpowiednio do pól opisujących dzień, miesiąc, rok.

Zad. 2. Zdefiniować klasę `Czas` zawierającą pola prywatne: `int godzina`, `int minuta`, `int sekunda`. Klasa powinna zawierać funkcje dostępowe, pobierające zawartość poszczególnych pól (np. `podajGodzine` itd.), oraz funkcje pozwalające na ustalanie zawartości pól (np. `ustawGodzine`, itd.). Klasa ma posiadać konstruktor domyślny zerujący wszystkie pola, oraz konstruktor ogólny, otrzymujący trzy parametry, trafiające odpowiednio do pól opisujących godzinę, minutę, sekundę.

Zad. 3. Dane jest równanie kwadratowe:

$$Ax^2 + Bx + C = 0$$

Należy zaprojektować klasę `RownanieKwadratowe`, przechowującą współczynniki takiego równania i pozwalającą na obliczanie wyznacznika (delty) oraz rozwiązań takiego równania. Klasa powinna posiadać prywatne pola A, B, C, będące liczbami całkowitymi, reprezentujące współczynniki równania. Każde z pól powinno posiadać akcesory w postaci funkcji składowych, np. `ustawA()`, `pobierzA()`. Klasa powinna posiadać funkcje składową `delta()` obliczającą wyróżnik trójmianu oraz trzy funkcje obliczające miejsca zerowe: jedną do obliczania pierwiastka podwójnego (dla delta==0) oraz dwie osobne, do obliczania pierwiastków, gdy delta ma wartość dodatnią. Obiekt ten może być wykorzystany w następujący sposób:

```
RownanieKwadratowe r;
double num;
cout << 'Podaj A:'; cin >> num; r.ustawA( num );
cout << 'Podaj B:'; cin >> num; r.ustawB( num );
cout << 'Podaj C:'; cin >> num; r.ustawC( num );
if( r.delta() > 0 )
  cout << "Pierwiastki rownania x1=" << r.obliczX1() << " x2="
  << r.obliczX2() << endl;
else
  if(r.delta() = 0 )
    cout << "Pierwiastek podwójny x12=" << r.obliczX12() << endl;
  else
    cout << "Brak pierwiastkow rzeczywistych" << endl;
```

Klasa powinna być wyposażona w konstruktor domyślny oraz trzyparametrowy konstruktor ogólny, co pozwoli deklarować obiekty klasy `RownanieKwadratowe` w następujący sposób:

```
RownanieKwadratowe r1;        // Konstruktor inicjuje A=B=C=0;
RownanieKwadratowe r2(3,2,1); // Konstruktor inicjuje A=3,B=2,C=1;
```

Należy się zastanowić nad przypadkiem, gdy równanie staje się liniowym, tzn. $A = 0$ i zaproponować rozwiązanie tego problemu.

Zad. 4. Dany jest układ równań liniowych:

$$A_1x + B_1y = C_1$$
$$A_2x + B_2y = C_2$$

Rozwiązanie układu równań może polegać na wyliczeniu odpowiednich wyznaczników W, W_x, W_y a następnie ich iloczynów — zgodnie z informacjami poznanym na zajęciach z matematyki. Należy zaprojektować i zaimplementować w języku C++ obiektowy program pozwalający na rozwiązywanie dowolnego układu takich równań. Program powinien umożliwiać wczytanie współczynników A_1, B_1, C_1, A_2, B_2, C_2, następnie powinien wyznaczyć rozwiązania równań metodą wyznacznikową. Należy identyfikować i prawidłowo zareagować na sytuację, gdy układ jest nieokreślony.

Zad. 5. Zdefiniować klasę `Punkt`, przeznaczoną do reprezentowania punków w przestrzeni dwuwymiarowej. Klasa powinna posiadać dwa pola prywatne x oraz y przeznaczone do przechowywania współrzędnych punktu, konstruktor domyślny, zerujący wartości tych pól, konstruktor ogólny umożliwiający ustawienie wartości pól (np. w deklaracji `Punkt p(10, 20);`). Ustawianie i pobieranie wartości pól powinny realizować funkcje dostępowe `ustawX`, `ustawY`, `pobierzX`, `pobierzY`.

Zad. 6. Zdefiniować klasy reprezentujące płaskie figury geometryczne `Kwadrat`, `Prostokat`, `Kolo`, `Trapez`. Klasy powinny posiadać pola przechowujące informacje niezbędne do obliczenia pól tych figur. Pola powinny być prywatne, dostęp do nich powinien być zapewniany przez odpowiednie metody pobierania i ustawiania wartości. Klasy powinny posiadać konstruktor bezparametrowy oraz ogólny, zapewniające inicjalizację wszystkich pól. Obliczenie pola danej figury powinna realizować odpowiednia funkcja składowa o nazwie `obliczPole` — wyznaczona wartość pola powinna być jej rezultatem.

Zad. 7. Bazując na klasie `Punkt` (zdefiniowanej w trakcie rozwiązywania zadania nr 5) oraz wykorzystując mechanizm dziedziczenia, zdefiniować klasę `Punkt3D`, reprezentującą punk w przestrzeni trójwymiarowej. Klasa ta powinna posiadać dodatkowe pole z przechowujące współrzędną trzeciego wymiaru, odpowiednie funkcje dostępowe `ustawZ`, `pobierzZ`, własny konstruktor bezparametrowy i parametrowy.

Zad. 8. Bazując na klasie `Data` (zdefiniowanej w trakcie rozwiązywania zadania nr 1) zdefiniować z wykorzystaniem dziedziczenia klasę `DataZKontrola` — w tej klasie należy dokonać redefinicji funkcji ustawiających wartości pól, tak by kontrolowały one poprawność wartości przekazywanych parametrów (roku, miesiąca, dnia).

Zad. 9. Bazując na klasie `Czas` (zdefiniowanej w trakcie rozwiązywania zadania nr 2) zdefiniować z wykorzystaniem dziedziczenia klasę `CzasZKontrola` — w tej klasie należy dokonać redefinicji funkcji ustawiających wartości pól, tak by kontrolowały one poprawność wartości przekazywanych parametrów (godziny, minuty, sekundy).

Zad. 10. Bazując na klasach opisu figur płaskich zdefiniowanych w zadaniu nr 6, zdefiniować ich klasy pochodne, reprezentujące

bryły: sześcian (pochodna klasy reprezentującej kwadrat), prostopadłościan (pochodna klasy reprezentującej prostokąt), kula (pochodna klasy reprezentującej koło), graniastosłup o podstawie trapezu (pochodna klasy reprezentującej trapez). W klasach pochodnych należy dodać wszelkie informacje konieczne dla obliczenia pól tych brył, oraz należy przedefiniować funkcje składowe obliczania pola (funkcje `obliczPole` każdej z klas reprezentujących figurę płaską), tak by wyznaczały właściwe pola brył.

Zad. 11. Zdefiniować klasę `Figura`, reprezentującą abstrakcyjną figurę geometryczną. Klasa powinna posiadać funkcję składową o nazwie `obliczPole`, której rezultatem ma być wartość 0, oraz funkcję `wypiszNazwe`, która wypisze do strumienia wyjściowego programu napis `Figura`. Następnie należy zmodyfikować definicje klas z zadania nr 6, tak by klasy te były klasami pochodnymi klasy `Figura`. W klasach pochodnych należy dokonać redefinicji funkcji `wypiszNazwe`, tak by wyprowadzała ona do strumienia wyjściowego nazwę każdej z klas. Zakłada się, że każda z klas dokonuje również redefinicji funkcji `obliczPole`, zgodnie z treścią zadania nr 6. Dziedziczenie oraz redefinicje funkcji składowych powinny być tak zrealizowane, by poprawnie działały następujące wywołania:

```
Kwadrat   kw( 2 );
Kolo      ko( 1 );
Prostokat pr( 3, 2 )
Figura *  f;
f = &kw;
f->wypiszNazwe();
cout << ' ' << f->obliczPole() << endl;  // Kwadrat 4
f = &ko;
f->wypiszNazwe();
cout << ' ' << f->obliczPole() << endl;  // Kolo 6.28318531
f = &pr;
f->wypiszNazwe();
cout << ' ' << f->obliczPole() << endl;  // Prostokat 6
```

Zad. 12. Zdefiniuj klasę `KolekcjaFigur`, pozwalającą na przechowywanie informacji o figurach zdefiniowanych w zadaniu nr 11. Klasa powinna posiadać prywatną tablicę wskaźników, w której będą zapisywane wskaźniki na figury dopisywane do kolekcji. Można założyć, że tablica będzie miała dostatecznie duży, statycznie określony rozmiar. Dopisanie figury do kolekcji powinna realizować funkcja `dodajFigure`, którą można wywołać następująco:

```
KolekcjaFigur kf;
kf.dodajfigurę( new Kwadrat );
kf.dodajfigurę( new Kolo );
kf.dodajfigurę( new Prostokat );
```

Klasa powinna posiadać funkcję `obliczSumarycznePole`, która powinna wyznaczać sumę pól wszystkich figur zapisanych w kolekcji. Operaja te powinna być wykonana poprzez wywołanie funkcji składowych `obliczPole` każdej figur z kolekcji, oraz zsumowanie ich rezultatów. Klasa powinna posiadać funkcję `wyczysc`, która sprawia, że kolekcja staję się pusta, jednak bez zwalniania jakiejkolwiek pamięci przydzielonej obiektom reprezentującym figury, oraz funkcję `kasuj`, która sprawia, że kolekcja staję się pusta, oraz zwalnia pamięć przydzieloną obiektom reprezentującym figury.

Java

Wprowadzenie

Język programowania Java jest językiem ogólnego przeznaczenia, z możliwością zrównoleglenia obliczeń, opartym na klasach, obiektowym. Został zaprojektowany na tyle prosto, że łatwo można osiągnąć biegłość w programowaniu w tym języku. Jest językiem pokrewnym C i C++, ale jest zorganizowany nieco inaczej — wiele aspektów C i C++ zostało pominiętych i włączono kilka pomysłów z innych języków. Z założenia miał to być język dla programistów, a nie naukowców i tak, jak C. A. R. Hoare zaproponował w swoim klasycznym artykule o tworzeniu języków programowania, unikano nowych i niesprawdzonych elementów.

Java jest językiem o silnym typowaniu. Pozwala to rozpoznać błędy kompilacji, które mogą i powinny być wykrywane w czasie kompilacji, oraz te, które występują w czasie wykonywania. Kompilacja w tym języku jest tłumaczeniem programów do maszynowo niezależnego kodu bajtowego (*byte-code*) w formacie zgodnym ze specyfikacją wirtualnej maszyny Javy. W momencie uruchomienia programu następuje załadowanie i łączenie klas potrzebnych do jego realizacji, opcjonalne generowanie kodu maszynowego, dynamiczna optymalizacja programu i wreszcie jego wykonanie.

Typy danych dzielą się na typy pierwotne i referencyjne. Typy pierwotne są definiowane tak, że ich reprezentacja jest taka sama na wszystkich komputerach i we wszystkich implementacjach. Są to różnych rozmiarów liczby całkowite w kodzie uzupełnieniowym do 2 (ZU2)[24], pojedynczej i podwójnej precyzji liczby zmiennoprzecinkowe, zgodne ze standardem IEEE 754, typ logiczny i typ znakowy, oparty na zestawie znaków Unicode.

Typy referencyjne są typami związanymi z klasami, interfejsami i tablicami. Typy referencyjne są implementowane za pomocą dynamicznie tworzonych obiektów, które są instancjami klas lub tablic. Do każdego obiektu może istnieć wiele referencji. Wszystkie obiekty (w tym tablice) wspierają metody klasy `Object`, która jest korzeniem drzewa hierarchii klas. Predefiniowana klasa `String` obsługuje łańcuchy znaków standardu Unicode. Istnieją klasy opakowujące wartości typów pierwotnych wewnątrz obiektów.

Zmienne są typowanymi miejscami w pamięci. Zmienna typu pierwotnego posiada wartość dokładnie takiego typu prymitywnego. Zmienna typu klasy może posiadać puste odwołanie (`null`) lub

[24] Dla przypomnienia: liczbę ujemną w takim zapisie uzyskujemy przez zanegowanie odpowiedniej liczby dodatniej i dodanie wartości jeden. Na przykład z liczby 12 zapisanej na jednym bajcie 12 = 00001100 liczbę −12 uzyskujemy następująco: 00001100 → 11110011 → 11110100 = −12. ZU2 jest obecnie najpopularniejszym sposobem zapisu liczb całkowitych w systemach cyfrowych. Jego popularność wynika z faktu, że operacje dodawania i odejmowania są w nim wykonywane tak samo jak dla liczb binarnych bez znaku. Z tego też powodu oszczędza się na kodach rozkazów procesora.

odwołanie do obiektu, którego typ jest typem klasy lub podklasy tej klasy. Zmienna typu interfejsu może zawierać puste odwołanie lub odwołanie do instancji jakiejkolwiek klasy, która implementuje interfejs. Zmienna typu tablicowego może przechowywać `null` lub odwołanie do tablicy. Zmienna klasy `Object` może przechowywać `null` lub odwołanie do każdego obiektu, klasy lub tablicy.

Konwersje zmieniają typ określony w czasie kompilacji, a niekiedy również wartość wyrażenia. Promocje numeryczne są stosowane w celu konwersji operandów numerycznego operatora do wspólnego typu, w którym operacja ta może być wykonana. Jest to czynność bezpieczna — rzutownie dla typów referencyjnych jest sprawdzane w trakcie wykonania w celu zapewnienia bezpieczeństwa.

Program jest zbiorem pakietów podobnych do modułów języka Modula. Składowymi pakietów są klasy, interfejsy i podpakiety. Pakiety są podzielone na jednostki kompilacji, które zawierają deklaracje typu i mogą zaimportować typy z innych pakietów, aby nadać im krótkie nazwy. Pakiety mają nazwy w hierarchicznej przestrzeni nazw, a system nazewnictwa domen internetowych zwykle może być używany do tworzenia unikalnych nazw pakietów.

Składowymi klas są klasy, interfejsy, pola (zmienne) i metody. Pola (zmienne) klasy są tworzone dla klasy i są wspólne dla wszystkich obiektów klasy. Metody klasy działają bez odniesienia do konkretnego obiektu — są wywoływane na rzecz klasy. Pola instancji są tworzone dla obiektów, które są instancjami klas. Metody instancji są wywoływane na rzecz obiektów.

W Javie obowiązuje pojedyncze dziedziczenie (brak jest dziedziczenia wielobazowego), w którym implementacja każdej klasy jest wyprowadzona z jednej klasy nadrzędnej, a ostatecznie z klasy `Object`. Zmienne typu klasy (obiekty) mogą odwoływać się do instancji tej klasy lub do jakiejkolwiek podklasy tej klasy, co umożliwia nowym typom używanie istniejących metod (polimorfizm).

Klasy wspierają programowanie współbieżne z synchronizowaniem metod. Metody deklarują rejestrowane wyjątki, które mogą wyniknąć z ich wykonania, co w trakcie kompilacji pozwala sprawdzić, czy zostaną one poprawnie obsłużone. Obiekty mogą zadeklarować metody finalizujące, które będą wywoływane przed usunięciem obiektów przez „odśmiecacz" (manadżer zarządzania nieużytkami — *garbage collector*), dzięki czemu obiekty mogą zwolnić swoje zasoby.

Dostęp do tablic uwzględnia kontrolę zakresu indeksów. Tablice są tworzonymi dynamicznie obiektami i mogą być przypisywane zmiennym typu `Object`. Język nie obsługuje tablic wielowymiarowych, ale tablice tablic.

Obsługa wyjątków jest w pełni zintegrowana z semantyką języka i mechanizmami współbieżności. Istnieją trzy rodzaje wyjątków: wyjątki rejestrowane, wyjątki czasu wykonania i wyjątki związane z błędami. Kompilator zakłada, że wyjątki rejestrowane są prawidłowo obsługiwane przez wymaganie, by metoda lub konstruktor mogła wyprowadzać wyjątek tylko wtedy, gdy taki wyjątek zadekla-

ruje. Dzięki temu na etapie kompilacji można stwierdzić, że istnieje obsługa takiego wyjątku. Większość wyjątków zdefiniowanych przez użytkownika to powinny być wyjątki rejestrowane. Z kolei nieprawidłowe operacje programu wykryte przez wirtualną maszynę Javy to wyjątki czasu wykonania (np. `NullPointerException`). Wreszcie wyjątki związane z błędami to reakcja na nieprawidłowości niezależne od programu, wykryte przez maszynę wirtualną (np. `OutOfMemoryError`). Większość prostych programów takich wyjątków nie powinna obsługiwać.

Większość instrukcji (oraz blok) bazuje na języku C. Java nie ma jednak instrukcji `goto`, ale z kolei posiada instrukcje `break` i `continue` z etykietą (obok ich podstawowej wersji). W przeciwieństwie do języka C, Java wymaga wyrażeń logicznych w instrukcjach sterujących (wartości całkowitych nie konwertuje na wartości boolowskie), co umożliwia wykrycie większej liczby błędów w czasie kompilacji. Synchronizacja na poziomie zasobu, metody lub obiektu stanowi podstawową realizację blokady w wielowątkowości. Za obsługę wyjątków odpowiada instrukcja `try` posiadająca sekcje `catch` oraz `finally`.

W kolejnych punktach zostały opisane główne elementy języka ze szczególnym uwzględnieniem klas. Dodatkowo, jako ilustrację zastosowania podejścia obiektowego, opisano także klasy opakowujące, tablice i łańcuchy znaków, jak również obsługę wyjątków.

Znakomita większość informacji została zaczerpnięta z dokumentacji języka Java dostępnej na stronach `http://docs.oracle.com/javase/`. Niezależnie od tego, na końcu skryptu podano przykładową literaturę, która może posłużyć do rozszerzenia wiedzy na temat tego języka.

Podstawy języka

Java jest językiem pokrewnym C i C++, ale jest zorganizowany nieco inaczej, między innymi wprowadzono szereg zmian i ograniczeń, np.:

- wprowadzono ograniczenia ułatwiające zrozumienie kodu np. nie można przysłaniać zmiennych,
- usunięto wskaźniki, uproszczono zarządzanie pamięcią (brak destruktorów, automatyczne odśmiecanie),
- usunięto preprocesor, niektóre konstrukcje np. `typedef`, instrukcję `goto`, typ `struct`, niektóre zmodyfikowano,
- usunięto możliwość przeciążania operatorów,
- usunięto możliwość wielokrotnego dziedziczenia wprost — wprowadzono implementację interfejsów,
- brak funkcji i zmiennych poza definicjami klas,

- wynikiem kompilacji jest kod pośredni (bajtowy), a nie maszynowy, jak w C++, dzięki czemu osiągnięto niezależność od platformy sprzętowej,
- „kompilator" generuje bezpieczny i objętościowo mały kod wynikowy, co jest zaletą w rozwiązaniach sieciowych.

Na środowisko uruchomieniowe języka Java składa się stosunkowo niewielkie jądro oraz zbiór pakietów pełniących funkcje bibliotek. Każdy pakiet zawiera zbiór powiązanych ze sobą klas. Zawartość bibliotek udostępniana jest za pomocą instrukcji `import`, np.:

```
import java.io.DataInputStream; // klasa strumienia wejściowego
import java.lang.Math.*;        // zasoby klasy "Math"
```

W przeciwieństwie do języka C++ (proceduralno-obiektowego), Java jest silnie ukierunkowana na obiektowość. Wszelkie dane i funkcje są pogrupowane w klasy. Program składa się więc ze zbioru klas, z których jedna jest klasą główną i musi posiadać funkcję:

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

od której wirtualna maszyna Javy rozpoczyna wykonywanie programu. Definicja klasy przedstawia się następująco[25]:

```
[⟨ModyfikatorKlasy⟩] class ⟨NazwaKlasy⟩
  [extends ⟨Nadklasa⟩]
  [implements ⟨Interfejsy⟩]
   {
       /* definicje metod i pól */
   }
```

[25] Nawiasy kwadratowe używane w opisie syntaktycznym elementów języka wskazują, co można opuścić.

Przykładowa klasa reprezentująca aplikację może wyglądać następująco:

```
public class Równanie {
  double a, b, c;                          // deklaracja pól
  double x1, x2;                           // deklaracja pól
  Równanie( double a_, double b_, double c_) // konstruktor
  {...}
  double Delta() {...}                     // metoda
  int Rozw() {...}                         // metoda
  public static void main(String[] args)   // metoda startowa
  {...}
}
```

Zmienne

Zmienna jest miejscem w pamięci z przyporządkowanym typem znanym w czasie kompilacji, który jest albo typem pierwotnym, albo referencyjnym. Zmienna zawsze posiada wartość zgodną (w sensie przypisania) z jej typem. Zmienna typu pierwotnego zawsze posiada wartość typu identycznego z własnym typem, zmienna typu klasy

`T` może przyjmować wartość `null` („pustą” referencję), referencję do instancji (obiektu) klasy `T` lub dowolnej podklasy klasy `T`. Jeśli `T` jest typem pierwotnym, wtedy zmienna tablicowa z elementami typu `T` może zawierać `null` lub referencję do tablicy z elementami typu `T`. Jeżeli `T` jest typu referencyjnego, zmienna typu tablicowego z elementami typu `T` może zawierać wartość `null` lub referencje do dowolnej tablicy z elementami typu `S`, gdzie typ `S` jest typem „przypisywalnym” do typu `T`. Zmiena typu `Object` może zawierać `null` lub referencję do dowolnego obiektu lub tablicy.

Wartość zmiennej jest zmieniana za pomocą operatora przypisania bądź operatora inkrementacji lub dekrementacji (prefiksowego lub postfiksowego).

Typy pierwotne

Pierwotne typy danych pokazano w tab. 1.

Tabela 1: Typy pierwotne Javy

Nazwa	**Zakres**	**Rozmiar**	**Domyślnie**
	całkowite		
`byte`	-128..127	1B	`(byte)`0
`short`	-32768..32767	2B	`(short)`0
`int`	-2147483648..2147483647	4B	0
`long`	-9223372036854775808.. 9223372036854775807	8B	0L
	rzeczywiste		
`float`	$\pm 1.40239846 \cdot 10^{-45}$– $\pm 3.40282347 \cdot 10^{38}$	4B (IEEE 754) 8–9 cyfr dokł.	0.0F
`double`	$\pm 4.94065645841246544 \cdot 10^{-324}$– $\pm 1.79769313486231570 \cdot 10^{308}$	8B (IEEE 754) 17–18 cyfr dokł.	0.0
	znakowy		
`char`	’\u0000’..’\uFFFF’	2B Unicode	’\u0000’
	logiczny		
`boolean`	`false..true`	nie określono precyzyjnie	
	„pusty”		
`void`			`false`

UWAGI:

- rozmiar typu jest stały i ściśle określony w specyfikacji języka (niezależny od platformy), stąd brak operatora `sizeof()`
- dopuszczalne konwersje automatyczne:
  `char` → `byte` → `short` → `int` → `long` → `float` → `double`

  w drugą stronę — konwersje jawne, np:

  `(int)`1.0

  `(long)`1.5 → 1 (odcięcie części ułamkowej)

  `(float)`3

  1D

3F

1234567890123456789L (L obowiązkowe — literał poza zakresem `int`)

(`int`)20L → wynik typu `int` (ważniejsze rzutowanie typu)

konwersje mogą zmienić wartość, np.

(`byte`)200 → -56

(`short`) 1234567890123456789L → -32491

Literały:

- logiczne (`boolean`):

  `false`, `true`

- całkowite:

  –dziesiętne:
  123, -342

  –ósemkowe:

0123	(83_{10})
0777	(511_{10})

  –szesnastkowe:

0x123	(291_{10})
0xABCD	(43981_{10})

- rzeczywiste:

1.0E-2	(domyślny typ double)
1.5F	(konieczna konwersja)
(`float`)2.35;	(konwersja konieczna)

- znakowe:

'1', '*', 'A'		
'\t'	(HT)	
'\b'	(BS)	
'\r'	(CR)	
'\s'	(FF)	
'\n'	(LF)	
'\''	(apostrof)	
'\"'	(cudzysłów)	
'\ó́ó'	(ó — cyfra ósemkowa)	np. '\101' → 'A'
'\uhhhh'	(h — cyfra szesnastkowa)	np. '\u0041' → 'A'

UWAGI:

- operacje na liczbach typu `byte` i `short` dają wynik typu `int` (domyślny typ dla literałów)
- typ `char` w operacjach zamieniany na `int`

Wyrażenia

Wyrażenie to konstrukcja (w postaci połączenia operatorów, wywołań funkcji, operandów i nawiasów) zwracająca wartość — wyrażenie jest reprezentacją tej właśnie wartości. Najprostszymi wyrażeniami są zmienne i stałe; bardziej złożone wyrażenia są budowane z wyrażeń prostych używając operatorów, funkcji, indeksowania, konwersji typów itd.

Przykładami wyrażeń mogą być:

```
x            // zmienna
15           // stała całkowita
Fun(x, y)    // wywołanie funkcji Fun
x * y        // iloczyn x i y
z / (1 − z)  // iloraz z i (1 − z)
x == 1.5     // wyrażenie logiczne
!done        // negacja wartości logicznej
(char)48     // konwersja typu
x = y        // przypisanie
```

Szczególnym rodzajem wyrażenia jest L-wartość (*l-value*), jednostka syntaktyczna dozwolona po lewej stronie operatora przypisania. Odnosi się ona do wartości, która istnieje dłużej niż przez jedno wyrażenie i może podlegać modyfikacjom, np:

```
x            // zmienna
tab[ 4 ]     // element tablicy
ob.pole      // pole obiektu
```

Jak wspomniano, wyrażenia są budowane między innymi z operatorów. W Javie operatory można podzielić na:

- prosty i złożony operator przypisania:

```
=      // proste przypisanie:        x=y
⟨op⟩=  // złożone przypisanie:       x+=y, a>>>=2
```

- operatory arytmetyczne:

```
+      // dodawanie (również składanie łańcuchów):
       // 11 + 22, "Ala" + " i As" → "Ala i As"
−      // odejmowanie
*      // mnożenie
/      // dzielenie:                 10.0 / 4.0 → 2.5, 10 / 4 → 2
%      // modulo:                    10 % 4 → 2, 10 % 4.1 → 1.8
```

- operatory jednoargumentowe:

```
+      // plus
−      // minus — zmienia wartość na przeciwną
++     // inkrementacja (pre lub post):  ++x, y++
−−     // dekrementacja (pre lub post):  −−x, y−−
!      // negacja wartości logicznej:    !false → true i odwrotnie
```

- operatory porównania i relacyjne:

```
==    // równe
! =   // różne
>     // większe
>=    // większe lub równe
<     // mniejsze
<=    // mniejsze lub równe
```

- operatory warunkowe:

```
&&    // logiczne AND
||    // logiczne OR
? :   // skrót dla if-then-else
```

- operator porównania typów:

```
instanceof // sprawdzenie zależności obiekt → klasa
```

- operatory bitowe i przesunięć bitowych:

```
~     // negacja bitów:                    ~0 → −1, ~ −128 → 127
<<    // przesunięcie logiczne w lewo: 33 << 3 → 264
>>    // przesunięcie arytmetyczne w prawo:
      // (byte)((byte)−128 >>>1) → −64
>>>   // przesunięcie logiczne w prawo:
      // 33 >> 3 → 4
&     // bitowe AND:                       12 & 22 → 4
^     // bitowe XOR:                       12 ^ 22 → 26
|     // bitowe OR:                        12 | 22 → 30
```

Wyróżnia się także dodatkowe operatory:

```
[ ]       // deklarowanie, tworzenie lub dostęp do elementu tablicy
.         // selektor pola lub metody
(⟨par⟩)   // lista parametrów funkcji
(⟨typ⟩)   // (⟨typ⟩)⟨wyrażenie⟩ — konwertuje wartość wyrażenia do
          // określonego typu
new       // utworzenie nowego obiektu lub tablicy
```

W czasie wartościowania wyrażeń ważną rolę odgrywają priorytety operatorów (tab. 2).

Instrukcje i bloki

Jak w każdym języku programowania, w Javie wporowdzono pewien podział instrukcji:

- wyrażeniowe:
  - wyrażenia z operatorami przypisania

		Typ	Operator
Ulcer	Jedzie	Jednoargumentowe	[ ] . ( ) ++(post) −−(post) + − ++(pre) −−(pre) ! ~ `new` (⟨typ⟩)
Addicts	Andrzej	Arytmetyczne	* / % + − << >> >>>
Really	Rowerem	Relacyjne	> < <= >= == ! =
Like	Ledwie	Ligiczne i bitowe	& ^ | && ||
C	Wlokąc	Warunkowy	? :
A lot	Pedałami	Przypisania	= += −= *= /= %= ...

Tabela 2: Operatory Javy (dla łatwiejszego zapamiętania wymyślono mnemoniki)

  - wyrażenia z operatorami ++ i −−
  - wywołania metod
  - tworzenie obiektów

- deklaracyjne
- sterujące:
  - warunkowe (decyzyjne):

    `if-else`

    `switch-case`
  - iteracyjne (pętle):

    `while`

    `do-while`

    `for`
  - obsługi wyjątków:

    `try-catch-finally`

    `throw`
  - skoki:

    `break`

    `continue`

    `return`
- blok // formalnie nie jest instrukcją, lecz pełni jej funkcję

Warto zauważyć brak w tym zbiorze instrukcji skoku `goto`. Mimo, że słowo kluczowe `goto` jest w Javie, jednak brak jego zastosowania (nie wolno go używać).

Instrukcje wyrażeniowe:

- wyrażenie zakończone średnikiem „ ; ”,
- tylko wyrażenia zmieniające wartość L-wyrażenia.

Przykłady:

```
wart=8933.234;                    // przypisanie
wart++;                           // postinkrementacja
--wart;                           // predekrementacja
System.out.println("Hello World!");   // wywołanie metody
mójRower = new tRower();          // utworzenie obiektu
```

Instrukcja deklaracyjna:
— deklaracja zmiennej z nadaniem wartości.
Przykłady:

```
int wart = 1234;                  // zmienna typu int
double wart = 8933.234;           // zmienna typu double
double delta = Delta();           // deklaracja z wywołaniem metody
tRower mójRower = new tRower();   // deklaracja z utworzeniem obiektu
```

Instrukcje sterujące przerywają liniowy przepływ sterowania wykorzystując do tego celu pętle oraz warunkowo wykonywane instrukcje lub bloki. Instrukcje te dokonują także przerwania pętli, funkcji oraz pozwalają na odpowiednią reakcję programu na sytuacje nadzwyczajne (wyjątki).

Podstawową instrukcją warunkową jest instrukcja `if-else` o tradycyjnej budowie:

```
if(⟨wyrażenieLogiczne⟩) ⟨instrukcjaLubBlok1⟩
  else ⟨instrukcjaLubBlok2⟩
```

lub bez frazy `else`:

```
if(⟨wyrażenieLogiczne⟩) ⟨instrukcjaLubBlok⟩
```

⟨wyrażenieLogiczne⟩ musi dawać wynik typu `boolean` i nie ma możliwości zastąpienia go wyrażeniem typu całkowitego (tak jak trzeba w C, a można w C++). ⟨instrukcjaLubBlok⟩ jest pojedynczą instrukcją lub ciągiem instrukcji zamkniętych w strukturze bloku (ograniczonym nawiasami „ { ” oraz „ } ”).

Drugą instrukcją warunkową jest `switch` wybierająca odpowiedni fragment kodu na podstawie wartości wyrażenia sterującego typu całkowitego w nowszych wersjach Javy także typu łańcuchowego `String`). Jej działanie jest identyczne jak w języku C/C++, więc zostanie tutaj pokazana tylko na kilku przykładach. W swej podstawowej postaci uruchamia ciąg instrukcji, począwszy od tej, której etykieta wyboru ma wartość równą wartości wyrażenia sterującego.

Chcąc wymusić działanie polegające na wykonaniu wykluczających się wyborów należy użyć instrukcji `break` (by przerwać działanie instrukcji `switch`) lub `return` (by jednocześnie opuścić blok funkcji zawierającej instrukcję `switch`):

```
public class Switch {
    static int CyfraHEX(char zn)
    { switch( zn )
        { case '0':
          case '1':
          case '2':
          case '3':
          case '4':
          case '5':
          case '6':
          case '7':
          case '8':
          case '9': return zn - '0';
          case 'a':
          case 'b':
          case 'c':
          case 'd':
          case 'e':
          case 'f': return zn - 'a' + 10;
          case 'A':
          case 'B':
          case 'C':
          case 'D':
          case 'E':
          case 'F': return zn - 'A' + 10;
          default: return -1;
        }
    }
    public static void main(String[] args)
    { System.out.println(CyfraHEX('0'));  // 0
      System.out.println(CyfraHEX('9'));  // 9
      System.out.println(CyfraHEX('a'));  // 10
      System.out.println(CyfraHEX('A'));  // 10
      System.out.println(CyfraHEX('f'));  // 15
      System.out.println(CyfraHEX('F'));  // 15
      System.out.println(CyfraHEX('*'));  // -1
    }
}
```

Należy dodać, że sekcja `default` jest opcjonalna, a jej umiejscowienie jest dowolne. Zawsze jednak zostanie ona uruchomiona dopiero, gdy nie zajdzie żaden jawnie przedstawiony przypadek, co prezentuje poniższy przykład:

```
switch( val )
  { case 3 : System.out.print("*");
    default: System.out.print("*");
    case 1 : System.out.println("*");
  }
```

```
switch( val )
  { case 3 : System.out.println("***");
             break;
    case 1 : System.out.println("*");
             break;
    default: System.out.println("**");
  }

// W obu przypadkach dla podanych wartości zmiennej val
// otrzymuje się wyniki:
//
// 1 --->  *
// 2 --->  **
// 3 --->  ***
```

Jak w wielu językach programowania, w Javie programista ma do dyspozycji trzy instrukcje iteracyjne. Najbardziej podstawową jest pętla `while`:

```
while(⟨wyrażenieLogiczne⟩)
  ⟨instrukcjaLubBlok⟩
```

gdzie ⟨wyrażenieLogiczne⟩daje wartość `false` lub `true` (tak jak poprzednio nie można używać wartości całkowitych), a ⟨instrukcjaLubBlok⟩ jest jak zwykle dowolną, pojedynczą instrukcją lub blokiem zamykającym dowolny ciąg instrukcji.

Instrukcja `do-while`

```
do
  ⟨instrukcjaLubBlok⟩
while(⟨wyrażenieLogiczne⟩)
```

Poszczególne składniki pełnią tutaj taką samą funkcję oraz podlegają takim samym ograniczeniom, jak w instrukcji `do`.

Instrukcja `for` jest jak zwykle pętlą typu `while` (sprawdzenie warunku przed wejściem do pętli). Jej postać jest następująca:

```
for(⟨Wi⟩; ⟨Wt⟩; ⟨Wz⟩) ⟨instrukcjaLubBlok⟩
```

gdzie ⟨Wi⟩ jest wyrażeniem inicjującym, ⟨Wt⟩ — wyrażeniem testującym, a ⟨Wz⟩ — wyrażeniem zwiększającym. Działanie pętli `for` jest następujące:

1. wykonaj czynności związane z wyrażeniem inicjującym ⟨Wi⟩,
2. wylicz wartość wyrażenia testującego ⟨Wt⟩; jeżeli wartość tego wyrażenia wynosi `false` to zakończ działanie pętli,
3. wykonaj instrukcję we wnętrzu pętli,
4. wylicz wartość wyrażenia zwiększającego ⟨Wz⟩,
5. przejdź do punktu 2.

Wyrażenie inicjujące może zawierać deklaracje zmiennych (z nadaniem wartości) i obok wyrażenia zwiększającego jest jedynym miejscem w programie, gdzie można używać znanego z języka C/C++ operatora „ , ” (przecinek) służącego do przetwarzania sekwencyjnego. Umożliwia to prostą implementację pętli z wieloma licznikami, np.:

```
int i, j;
for(i=1, j=11; i*j<30; i++, j-- )  // pętla z dwoma licznikami
  System.out.println( "i=" + i + "j=" + j );
// i=1, j=11
// i=2, j=10
// i=3, j=9
```

Zmienne zadeklarowane w wyrażeniu ⟨Wi⟩ są widoczne tylko we wnętrzu instrukcji `for` — po wykonaniu pętli nie mogą być one ponownie użyte.

Wszystkie trzy wyrażenia w nagłówku pętli `for` są opcjonalne, a forma:

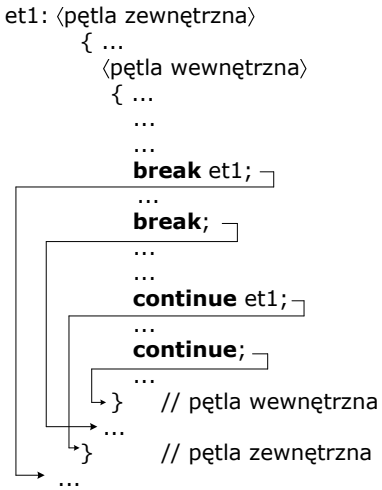
```
for ( ; ; ) ⟨instrukcjaLubBlok⟩
```

oznacza pętlę nieskończoną, która powinna być przerwana za pomocą odpowiedniej instrukcji wewnętrznej.

Pętla `for` ma także drugą, rozszerzoną formę przeznaczoną do operowania na kolekcjach oraz tablicach. Czyni ona te instrukcję bardziej zwięzłą i czytelną dla tego zastosowania. Można tę postać przedstawić na następującym przykładzie:

```
public static void main(String[] args)
  { int[] tabLiczb = { 1, 3, 5, 7, 9 };
    for (int numer: tabLiczb)
    { System.out.println("Liczba: " + numer );
    }
  }
```

Wykonaniem każdej pętli można sterować za pomocą instrukcji `break` oraz `continue`. Pierwsza z nich przerywa bieżącą pętlę (powoduje opuszczenie jej zakresu), a druga kończy bieżącą iterację; dla pętli `while` oraz `do` następuje przejście do sprawdzenia warunku kontynuowania pętli, dla pętli `for` — do wyrażenia zwiększającego. Nie można jednak za ich pomocą przerwać działania pętli zagnieżdżonych. Jak wspomniano wcześniej, brak w Javie instrukcji `goto`, jednak jej funkcję częściowo przejęły instrukcje `break` oraz `continue` w wersji z etykietą, przy czym etykietuje się instrukcję pętli, której będzie dotyczyło użycie wspomnianych instrukcji przerywających (rys. 29).



Rysunek 29: Instrukcje `break` i `continue` w pętlach

Programowanie obiektowe w Javie

Java jest językiem w pełni obiektowym — klasa jest postawowym budulcem programów. W Javie zaprojektowano charakterystyczne

elementy programowania obiektowego: hermetyzację, dziedziczenie, polimorfizm. W porównaniu z językiem C++ zrezygnowano z przeciążania operatorów oraz z dziedziczenia wielobazowego na rzecz implementacji interfejsów. Dodatkowo wszystkie metody są domyślnie wirtualne. Koncepcję klas i obiektów rozszerzono o tablice i łańcuchy znaków. Za pomocą podejścia obiektowego zrealizowane są również takie aspekty programowania, jak obsługa wyjątków czy wielowątkowość.

Klasy i obiekty

Podstawowe zasady tworzenia i wykorzystywania klas w Javie można przedstawić następująco:

- program jest zbiorem klas,
- wszystkie klasy dziedziczą z klasy `Object` Javy, nawet, gdy nie zaznaczono tego w definicji klasy,
- domyślnie klasy są zawsze dostępne dla wszystkich innych klas zawartych w tym samym pakiecie,
- w pliku źródłowym może wystąpić tylko jedna klasa publiczna o nazwie identycznej z nazwą pliku,
- obiekty klas po zadeklarowaniu muszą zostać utworzone za pomocą operatora `new` z konstruktorem klasy (lub za pomocą inicjatora),
- obliczenia w klasie rozpoczynają się (jeśli jest) od statycznego wyrażenia (bloku) inicjującego.

  W Javie definicja klasy jest zbudowana w sposób opisany poniżej:

```
[⟨ModyfikatorKlasy⟩] class ⟨NazwaKlasy⟩
  [extends ⟨Nadklasa⟩]
  [implements ⟨Interfejsy⟩]
   {
       /* definicje metod i pól */
   }
```

Definicja rozpoczyna się od opcjonalnego modyfikatora klasy, który może być dozwoloną kombinacją modyfikatora dostępu i modyfikatora właściwości:

- modyfikatory dostępu:
  - `public` — specyfikator dostępu określający nieograniczoną dostępność klasy — jest ona widoczna z zewnątrz pakietu, w którym się znajduje,
  - ⟨`puste`⟩ — domyślnie klasa jest dostępna jedynie dla wszystkich klas zdefiniowanych wewnątrz pakietu, w którym się znajduje; poza nim jest niewidoczna;

- modyfikatory właściwości:
  - `abstract` — definiuje klasę abstrakcyjną; nie jest możliwe utworzenie żadnych obiektów tej klasy, jest ona użyteczna jedynie podczas budowania hierarchii klas (przy dziedziczeniu); klasa abstrakcyjna powinna (choć formalnie nie musi) zawierać przynajmniej jedną metodę abstrakcyjną (zadeklarowaną z modyfikatorem `abstract` i niezaimplementowaną); jeżeli klasa zawiera choć jedną metodę abstrakcyjną, wtedy musi być zadeklarowana jako abstrakcyjna,
  - `final` — klasa finalna — od klasy tego typu niemożliwe jest utworzenie klas pochodnych; stosuje się je w celu zapewnienia odpowiedniego poziomu bezpieczeństwa aplikacjom oraz niekiedy do oznaczenia klas, nad którymi prace zostały ukończone.

Po nazwie klasy następuje opcjonalna deklaracja klasy bazowej (jednej — Java nie dopuszcza dziedziczenia wieolobazowego). Następnie można umieścić listę nazw implementowanych interfejsów rozdzielonych przecinkami.

We wnętrzu klasy znajduje się dowolna liczba deklaracji pól danych, definicji metod i klas zagnieżdżonych; deklaracje i definicje mogą być umieszczone w dowolnej kolejności, choć zalecane jest by rozpoczynać od deklaracji pól.

Pola

Pola danych są atrybutami klasy, pełniącymi rolę podobną do zmiennych lub stałych. Są one deklarowane na tych samych zasadach, co zmienne lokalne:

```
[⟨ModyfikatorPola⟩] ⟨TypWartości⟩ ⟨NazwaPola⟩ [=⟨Inicjator⟩];
```

Do deklarowania pól również używa się modyfikatorów dostępu i właściwości, w szczególności:

- modyfikatory dostępu:
  - `public` — pola zadeklarowane jako publiczne są dostępne z zewnątrz bez ograniczeń,
  - `private` — pola zadeklarowane jako prywatne dostępne są tylko dla metod klasy właściciela,
  - `protected` — dostęp do pól zabezpieczonych dozwolony jest dla metod klasy właściciela i klas pochodnych (oraz dla klas w tym samym pakiecie);

  domyślnie pola są publiczne w ramach pakietu,
- modyfikatory właściwości:
  - `static`,
  - `final`,

– `volatile`.

– `transient`.

Pola zadeklarowane bez modyfikatora `static` są polami (zmiennymi) obiektu, pola statyczne są polami (zmiennymi) klasy. Pola obiektu są dostępne z poziomu obiektu (`obiekt.pole`), są tworzone w momencie tworzenia obiektu i inicjowane domyślnymi wartości odpowiednich typów (chyba, że są inicjowane jawnie). Pola statyczne są wspólne dla wszystkich obiektów klasy i dostępne zarówno z poziomu klasy jak i obiektów tej klasy. Są one tworzone w momencie ładowania klasy i odpowiednio inicjowane.

Pola oznaczone modyfikatorem `final` mogą być zainicjowane dokładnie jeden raz (niekoniecznie w momencie deklarowania) i nie mogą podlegać modyfikacjom, są odpowiednikiem stałych (w szczególności w połączeniu z modyfikatorem `static`)), np.:

```
static final double PI = 3.141592653589793;
```

Jednak dla finalnych pól typu klasy można zmieniać wartość pól składowych, np.:

```
class Klasa
{ int   p1=99;
  final int p2;
  { p2 = 1000;} // blok inicjujący

  public void Main()
    { final int   p3;
      final Klasa c = new Klasa();

      System.out.println(++c.p1);  // 100
      System.out.println(p3=1000); // 1000
    }
  public static void main(String[] args)
    { Klasa o = new Klasa();
      o.Main();
      System.out.println(o.p2);    // 10000
    }
}
```

Modyfikator `volatile` informuje kompilator, że pole może zostać zmodyfikowane asynchronicznie, przez konkurencyjne wątki (w programach wielowątkowych), a modyfikator `transient` powoduje, że pole jest pomijane w procesie serializacji[26] obiektu.

[26] W kontekście programowania komputerów jest to proces przekształcania obiektów w strumień bajtów, z zachowaniem aktualnego stanu obiektu. Serializowany obiekt może zostać utrwalony w pliku dyskowym, przesłany do innego procesu lub innego komputera poprzez sieć.

Metody

Metody opisują czynności wykonywane na danych obiektu lub klasy. Podstawowe cechy metod Javy można określić następująco:

- wszystkie metody muszą być definiowane w treści klas (z wyjątkiem metod abstrakcyjnych),

- domyślnie wszystkie metody są wirtualne (z wyjątkiem metod finalnych i statycznych),
- jeśli metoda jako wynik zwraca obiekt odpowiedniej klasy, to można bezpośrednio dopisać kolejne wywołanie metody:

```
wynik = obiekt.Metoda1().Metoda2();
```

- metody mogą być przeciążane: muszą się różnić liczbą lub/i typami parametrów (inaczej niż w C++, gdzie wystarczy, że zwracają wyniki różnych typów),
- w każdej metodzie niestatycznej jest dostępny obiekt, na rzecz którego ta metoda została wywołana, w postaci zmiennej `this`,
- przekazywanie parametrów:
  - parametry typów prostych są przekazywane przez wartość,
  - obiekty są przekazywane przez referencję, jednak kiedy obiekt zostaje w metodzie zmieniony na inny, to jest to widoczne jedynie w treści tej metody (nowy obiekt jest obiektem lokalnym):

```
void Test( Klasa ob2 )
{ ob2 = new Klasa();
  ob2.pole = W2;
}
...
  ob1 = new Klasa();
  ob1.pole = W1;
  Test( ob1 );
  // ob1.pole == W1
```

Składnia definicji metody jest następująca:

```
[⟨ModyfikatorMetody⟩] ⟨TypZwracanejWartości⟩ ⟨NazwaMetody⟩
    ([⟨Typ1⟩ ⟨NazwaParametru1⟩, ⟨Typ2⟩ ⟨NazwaParametru2⟩, itd.])
    [throws ⟨ListaWyjątków⟩]
  {
    /* implementacja (kod źródłowy) metody */
  }
```

Dodatkowo, ostatni parametr może mieć postać:

```
⟨Typ⟩... ⟨NazwaParametru1⟩
```

co oznacza, że w wywołaniu można umieścić dowolną liczbę parametrów aktualnych. Nazwa metody oraz jej lista parametrów tworzy sygnaturę metody.

Kwalifikatory metod, podobnie jak w przypadku pól, określają dostępność metod oraz ich dodatkowe właściwości. Modyfikatory dostępu do metod są takie same, jak modyfikatory dostępu do pól. Modyfikatory właściwości są nieco inne:

- `static`,
- `final`,
- `abstract`,
- `synchronized`,
- `native`.

Metody statyczne są nazywane metodami klasy i, podobnie jak w przypadku pól, można je wywoływać zarówno na rzecz obiektu jak i klasy; są one głównie używane do dostępu do pól statycznych. Metody niestatyczne są nazywane metodami obiektów.

Nie wszystkie kombinacje rodzajów pól i metod są dozwolone:

- metody obiektu mają bezpośredni dostęp do pól i metod obiektu,
- metody obiektu mają bezpośredni dostęp do pól i metod klasy,
- metody klasy mają bezpośredni dostęp do pól i metod klasy,
- metody klasy nie mają bezpośredniego dostępu do pól i metod obiektu — muszą otrzymać referencję do obiektu (np. za pomocą parametru); metody klasy nie mogą również używać zmiennej `this`, jako że w treści takich klas nie istnieje obiekt, którego zmienna `this` jest reprezentantem.

Kolejnym modyfikatorem właściwości jest modyfikator `final`. Metoda finalna nie może być przesłonięta przez metodę o tej samej sygnaturze w klasie potomnej; w szczególności metody wywoływane z konstruktora powinny być tak zadeklarowane, aby w podklasie nie nastąpiło wywołanie metody z podklasy (zamiast metody z klasy bazowej) ze względu na polimorfizm, który jest aktywowany przed rozpoczęciem obliczeń w konstruktorze:

```
class Bazowa
{ Bazowa()
    { Metoda();
    }
  void Metoda()
    { System.out.println("Bazowa");
    }
}
class Potomna extends Bazowa
{ Potomna()
    { Metoda();
    }
  void Metoda()
    { System.out.println("Potomna");
    }
  public static void main(String[] args)
    { Bazowa  obBaz = new Bazowa();    // "Bazowa"
      Potomna obPot = new Potomna();   // "Potomna"
```

```
                                        // "Potomna"
    }
}
```

Tworząc obiekt klasy `Bazowa` konstruktor tej klasy wywołuje metodę `Metoda()` z klasy `Bazowa`. Z kolei tworząc obiekt klasy `Potomna` konstruktor tej klasy najpierw wywołuje konstruktor klasy `Bazowa`. Ponieważ `Metoda()` klasy `Bazowa` nie jest finalna, z konstruktora klasy `Bazowa` zostanie wywołana `Metoda()` zdefiniowana w klasie `Potomna`, gdyż wszystkie te metody są wywoływane na rzecz obiektu klasy `Potomna`.

Metoda zadeklarowana z modyfikatorem `synchronized` na początku wykonywania blokuje dostęp do obiektu, do którego należy i odblokowuje go, gdy zakończy działanie. Jeśli dostęp do obiektu już wcześniej został zablokowany, to metoda taka oczekuje na odblokowanie obiektu zanim zacznie się wykonywać. Mechanizm ten ma bardzo istotne znaczenie w przypadku programów wielowątkowych, gdyż nie dopuszcza do jednoczesnego dostępu do danych przez wiele metod, co mogłoby prowadzić do uzyskania błędnych wyników.

Metody określone jako `native` są implementowane w języku innym niż Java (np. C++); oznacza to funkcje implementowane z wykorzystaniem nieprzenośnych cech danej platformy.

Konstruktory

Konstruktor jest metodą wykonywaną w momencie tworzenia obiektu. Nazwa konstruktora jest identyczna z nazwą klasy, ale nie podaje się dla niego typu zwracanej wartości (można przyjąć, że zwraca referencję do właśnie utworzonego obiektu danej klasy); konstruktory można przeciążać — muszą mieć różne sygnatury. Każda klasa posiada przynajmniej jeden konstruktor — jeśli nie zdefiniowano jawnie konstruktora, to jest tworzony bezparametrowy konstruktor domyślny. Każdy konstruktor powinien rozpoczynać działanie od wywołania konstruktora klasy bazowej za pomocą słowa kluczowego `super` (z ewentualnymi parametrami). Jeżeli tego nie wykona jawnie, wtedy automatycznie jest wywoływany bezparametrowy konstruktor klasy bazowej. W przypadku braku takiego konstruktora jest sygnalizowany błąd. Należy pamiętać, że jeżeli w klasie zdefiniowano choć jeden konstruktor, to konstruktor domyślny (bezparametrowy) nie jest tworzony.

W konstruktorze można wywołać inny konstruktor bieżącej klasy, ale nie należy zapętlać tych odwołań.

Przykłady konstruktorów:

```
class Liczba extends KlBazowa
{ Liczba()     // konstruktor bezparametrowy
    { super(); // jawne wywołanie bezparametrowego
    }          // konstruktora klasy bazowej
  Liczba(int l)
```

```
    {          // domyślne wywołanie bezparametrowego
    }          // konstruktora klasy bazowej
  Liczba( double l, int r )
    { super(l, r);
    }
  Liczba( int r, double k )
    { this(r); // wywołanie konstruktora bieżącej klasy
    }
   ...
}
```

Blok inicjujący

Blok inicjujący jest blokiem zawierającym instrukcje wykonywane w momencie inicjowania obiektów klasy (w momencie wywołania konstruktora), dla każdego obiektu dokładnie jeden raz, nawet gdy konstruktor wywołuje inny konstruktor. Jeśli blok inicjujący zgłasza wyjątek, to musi on być zadeklarowany w klauzuli `throws` każdego konstruktora. Klasa może posiadać wiele bloków inicjujących, są one wtedy wykonywane w kolejności ich położenia. Blok inicjujący ma dostęp do wszystkich zasobów klasy (statycznych i niestatycznych).

Można także tworzyć statyczne bloki inicjujące. Taki blok jest wtedy poprzedzony słowem `static`. Statyczny blok inicjujący jest wykonywany w czasie ładowania klasy, przed wykonaniem jakiegokolwiek konstruktora (również wtedy, gdy żaden konstruktor nie zostanie wykonany), niezależnie od tego, czy jest umieszczony przed, czy po niestatycznym bloku inicjującym. Statyczny blok inicjujący ma dostęp jedynie do statycznych zasobów klasy, gdyż jest wykonywany przed utworzeniem jakiegokolwiek obiektu danej klasy.

```
class Klasa {
  { System.out.println("Blok niestatyczny");
  }
  static{ System.out.println("Blok statyczny");
        }

  public static void main(String[] args)
    { Klasa ob = new Klasa();

      System.out.println("Obliczenia");
    }
}
// Blok statyczny
// Blok niestatyczny
// Obliczenia
/****** bez tworzenia obiektu klasy ,,Klasa'': ******/
// Blok statyczny
// Obliczenia
```

Interfejsy

W Javie interfejs jest typem referencyjnym podobnym do klasy, który zawiera jedynie stałe, deklaracje metod (metody bez treści) oraz zagnieżdżone typy danych. Interfejsy nie mogą służyć do tworzenia obiektów — mogą jedynie być implementowane przez klasy, bądź rozszerzane przez inne interfejsy. Definicja interfejsu jest podobna do definicji klasy:

```
[⟨ModyfikatorInterfejsu⟩] interface ⟨NazwaInterfejsu⟩
  [extends ⟨Interfejsy⟩]
   {
       /* definicje stałych */
   /* deklaracje metod */
   }
```

```
public interface OperateCar {
  // constant declarations, if any
  // method signatures
  int turn(Direction direction,   // An enum with values RIGHT, LEFT
            double radius, double startSpeed, double endSpeed);
  int changeLanes(Direction direction, double startSpeed, double endSpeed);
  int signalTurn(Direction direction, boolean signalOn);
  int getRadarFront(double distanceToCar, double speedOfCar);
  int getRadarRear(double distanceToCar, double speedOfCar);
        ......
  // more method signatures
}
```

Należy zauważyć, że deklaracja metody nie posiada nawiasów klamrowych (brak jest treści metody) i jest zakończona średnikiem.

Użycie interfejsu polega na napisaniu klasy implementującej ten interfejs. Gdy klasa (nieabstrakcyjna) implementuje interfejs, musi zdefiniować treść każdej metody zadeklarowanej w interfejsie.

```
public class OperateBMW760i implements OperateCar {
  // the OperateCar method signatures, with implementation --
  // for example:
  int signalTurn(Direction direction, boolean signalOn) {
    //code to turn BMW's LEFT turn indicator lights on
    //code to turn BMW's LEFT turn indicator lights off
    //code to turn BMW's RIGHT turn indicator lights on
    //code to turn BMW's RIGHT turn indicator lights off
  }
  // other members, as needed -- for example, helper classes
  // not visible to clients of the interface
}
```

W powyższym przykładzie interfejs `OperateCar` został zaimplementowany przez klasę `OperateBMW760i`. Implementacja dla samochodu Chevrolet czy Toyota będzie odmienna, lecz obie będą związane z tym samym interfejsem. Będą zawierały dane GPS,

cyfrowe mapy oraz dane dotyczące ruchu na drodze służące do kierowania ruchem samochodu. W tym celu będą tam zaimplementowane metody: `turn`, `brake`, `accelerate` zadeklarowane w interfejsie.

Interfejsy odgrywają jeszcze inną ważną rolę. Nie są częścią hierarchii klas, choć współpracują z klasami. Są w ten sposób alternatywą dla zakazu dziedziczenia wielokrotnego (wielobazowego). Wobec tego obiekty mogą posiadać wiele typów: typ pochodzący z klasy, do której należy oraz typy wszystkich interfejsów, które implementuje. Oznacza to, że jeżeli zmienna jest zadeklarowana jako typu interfejsu, jej wartość może się odnosić do dowolnego obiektu, który jest instancją dowolnej klasy implementującej dany interfejs.

Dziedziczenie

Domyślnie każda klasa w języku Java dziedziczy po klasie `Object`. Dlatego też wszystkie klasy posiadają metody takie jak:

- clone()
- equals()
- finalize()
- getClass()
- hashCode()
- notify()
- notifyAll()
- toString()
- wait()

Są to metody zaimplementowane w klasie Object. Samemu również można dziedziczyć po dowolnej klasie, która nie posiada modyfikatora final przed słowem kluczowym `class`.

Zagnieżdżanie klas

Java pozwala definiować klasę wewnątrz innej klasy. Taka klasa jest klasą zagnieżdżoną:

```
class OuterClass {
  ...
  class NestedClass {
    ...
  }
}
```

Zagnieżdżone klasy dzielą się na dwie kategorie: statyczne i niestatyczne. Statyczne klasy zagnieżdżone są zwykle w skrócie nazywane klasami zagnieżdżonymi, klasy niestatyczne są klasami wewnętrznymi.

```
class OuterClass {
  ...
  static class StaticNestedClass {
    ...
  }
  class InnerClass {
    ...
  }
}
```

Klasa zagnieżdżona jest składową zawierającej ją klasy. Klasy wewnętrzne (niestatyczne zagnieżdżone klasy) mają dostęp do innych składowych klasy właściciela, również do prywatnych. Statyczne klasy zagnieżdżone nie mają takiego dostępu. Klasy zagnieżdżone (statyczne i niestatyczne) mogą być deklarowane jako prywatne, zabezpieczone, publiczne lub prywatne dla pakietu.

Jest kilka przyczyn sugerujących stosowanie takich klas:

- jest to sposób logicznego grupowania klas, które są używane tylko w jednym miejscu,
- takie rozwiązanie wzmacnia hermetyzację,
- klasy zagnieżdżone zapewniają bardziej czytelny i ścisły kod.

Logiczne grupowanie klas — jeżeli klasa jest użyteczna tylko dla jednej innej klasy, wtedy jest to rozsądne, aby osadzić ją w tej klasie, aby trzymać obie klasy razem, przez co uzyskuje się większą prostotę budowy pakietu.

Wzmocnienie hermetyzacji — można rozpatrzeć dwie klasy najwyższego poziomu, A i B, gdzie klasa B musi mieć dostęp do składowych klasy A, które muszą być prywatne. Ukrywając klasę B w klasie A, składowe klasy A mogą być zadeklarowane jako prywatne i klasa B ma wciąż do nich dostęp. Dodatkowo klasa B również jest prywatna — niedostępna z zewnątrz.

Bardziej czytelny kod — zagnieżdżając małe klasy w klasach wyższego poziomu powoduje umieszczenie kodu blisko miejsca jego użycia.

Statyczne klasy zagnieżdżone

Tak jak w przypadku pól i metod klasy, statyczna klasa zagnieżdżona jest związana z jej klasą zewnętrzną i tak jak metody statyczne, statyczne klasy zagnieżdżone nie mogą się bezpośrednio odnosić do niestatycznych zmiennych i metod klasy zewnętrznej — mogą one być dostępne jedynie poprzez odpowiednie obiekty.

Należy zauważyć, że statyczna klasa zagnieżdżona współpracuje z niestatycznymi składowymi jej klasy zewnętrznej (a także innych klas) tak jak klasy najwyższego poziomu. W efekcie statyczne klasa zagnieżdżona jest od strony zachowania klasą najwyższego poziomu, która została zagnieżdżona w innej klasie najwyższeego poziomu dla uproszczenia struktury danego pakietu.

Statyczne klasy zagnieżdżone są dostępne za pomocą nazwy klasy zewnętrznej:

```
OuterClass.StaticNestedClass
```

Na przykład, aby utworzyć obiekt statycznej klasy zagnieżdżonej, należy użyć zapisu:

```
OuterClass.StaticNestedClass nestedObject =
  new OuterClass.StaticNestedClass();
```

Klasy wewnętrzne

Tak jak w przypadku pól i metod niestatycznych klasa wewnętrzna jest związana z obiektem jej zewnętrznej klasy i ma bezpośredni dostęp do pól i metod tego obiektu. Jednak, jako że klasa wewnętrzna jest związana z obiektem, nie może definiować żadnych składowych statycznych.

Obiekty będące instancją klasy wewnętrznej istnieją w obiektach klasy zewnętrznej, np. dla układu klas:

```
class OuterClass {
  ...
  class InnerClass {
    ...
  }
}
```

Obiekt klasy wewnętrznej może istnieć jedynie wewnątrz obiektu klasy zewnętrznej i ma bezpośredni dostęp do pól i metod obiektu, w którego wnętrzu został utworzony. Aby utworzyć obiekt klasy wewnętrznej należy najpierw utworzyć obiekt klasy zewnętrznej. Następnie w tym obiekcie zewnętrznym można utworzyć obiekt wewnętrzny:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Dodatkowo, można wyróżnić dwa specjalne rodzaje klas wewnętrznych: lokalne i anonimowe (również zwane anonimowymi klasami wewnętrznymi).

Poniżej zostanie przedstawiony przykład klasy wewnętrznej. W przykładzie zostanie rozważona tablica zapełniona liczbami całkowitymi, z której zostaną wyprowadzone wartości elementów o parzystych indeksach (w kolejności rosnącej).

Przykład składa się z:

1. Klasy zewnętrznej (`DataStructure`), która zawiera metody: dodającą wartość całkowitą do tablicy oraz wyprowadzającą wartości elementów o parzystych indeksach.

2. Klasy wewnętrznej (`InnerEvenIterator`), która jest bliska standardowemu iteratorowi Javy. Iteratory są używane do przechodzenia przez strukturę danych (lista, kontener) i zwykle mają metody służące do sprawdzania końca struktury (czy bieżący

element jest ostatnim), pobierania bieżącego elementu oraz przechodzenia do elementu następnego.

3. Głównej metody (`main`), która tworzy obiekt klasy `DataStructure` (`ds`) i używa go do wypełnienia tablicy (`arrayOfInts`) wartościami całkowitymi; następnie metoda `main` wywołuje metodę `printEven` wyprowadzającą wartości elementów o parzystych wartościach indeksów.

```
public class DataStructure {
  //create an array
  private final static int SIZE = 15;
  private int[] arrayOfInts = new int[SIZE];
  public DataStructure() {
    //fill the array with ascending integer values
    for (int i = 0; i < SIZE; i++) {
      arrayOfInts[i] = i;
    }
  }
  public void printEven() {
    //print out values of even indices of the array
    InnerEvenIterator iterator = this.new InnerEvenIterator();
    while (iterator.hasNext()) {
      System.out.println(iterator.getNext() + " ");
    }
  }

  //inner class implements the Iterator pattern
  private class InnerEvenIterator {
    //start stepping through the array from the beginning
    private int next = 0;
    public boolean hasNext() {
      //check if a current element is the last in the array
      return (next <= SIZE - 1);
    }
    public int getNext() {
      //record a value of an even index of the array
      int retValue = arrayOfInts[next];
      //get the next even element
      next += 2;
      return retValue;
    }
  }

  public static void main(String s[]) {
    //fill the array with integer values and print out
    //only values of even indices
    DataStructure ds = new DataStructure();
    ds.printEven();
  }
```

```
}
// 0 2 4 6 8 10 12 14
```

Należy przy tym zauważyć, że klasa `InnerEvenIterator` odwołuje się bezpośrednio do pola `arrayOfInts` obiektu klasy `DataStructure`.

Wewnętrzne klasy lokalne i anonimowe, modyfikatory

Java wprowadza dwa dodatkowe typy klas wewnętrznych. Można zdefiniować klasę wewnętrzną w treści metody. Taka klasa jest lokalną klasą wewnętrzną. Można również zdefiniować taką klasę bez nadania jej nazwy, wtedy klasa jest anonimową klasą wewnętrzną.

Do wszystkich rodzajów klas wewnętrznych (z wyjątkiem anonimowych) można używać modyfikatorów (`private`, `protected`, `public`), jak dla składowych klasy zewnętrznej.

Klasy opakowujące

Klasy opakowujące stanowią obiektowe wersje typów podstawowych. Motywacją są tutaj sytuacje, w których wygodniejsze lub konieczne staje się użycie obiektów w miejsce wartości typów prostych, m. in.:

- wygodne operacje za pomocą metod,
- zamiast funkcji „standardowych" — statyczne metody klas (m. in. do wykonywania różnych konwersji),
- reprezentacja obiektowa wymagana w niektórych zastosowaniach (np. listy, tablice haszujące).

Na przykład typy opakowujące dla typów liczbowych są przydatne gdy:

- argument pewnej metody powinien być obiektem (np. przy operacjach na kolekcjach),
- należy użyć stałych określających dolne lub górne ograniczenie zakresu wartości danej klasy (`MIN_VALUE` i `MAX_VALUE`),
- potrzebna jest konwersja wartości na lub z innego typu podstawowego, na lub z łańcucha znaków, konwersja pomiędzy różnymi systemami liczenia itp.

  Java udostępnnia klasy opakowujące:

  `Byte`,

  `Short`,

  `Integer`,

  `Long`,

  `Float`,

  `Double`,

```
Boolean,

Character,

Void,
```

oraz dodatkowe klasy „uzupełniające” dla typów liczbowych: `BigDecimal` oraz `BigInteger`.

Klasy opakowujące (z wyjątkiem klasy `Void`) posiadają wiele elementów wspólnych, między innymi:

- konstruktor z jednym parametrem typu pierwotnego — tworzy on obiekt odpowiedniej klasy, np:

  ```
  Integer(int i)
  Character(char c)
  ```

- konstruktor zamieniający napis na wartość typu klasy tworząc obiekt odpowiedniego typu (z wyjątkiem klasy `Character`), np.:

  ```
  Float(String s)
  Long (String s)
  ```

- metoda `toString()` zamieniająca pamiętaną w obiekcie wartość typu pierwotnego na łańcuch oraz metoda `toString(`⟨`typ pierwotny`⟩ `w)` zamieniająca na łańcuch wartość typu pierwotnego przekazaną przez parametr, np.:

  ```
  integer.toString()    // integer - obiekt klasy Integer
  Integer.toString(999) // metoda statyczna
  ```

- metody:

  – ⟨`typ pierwotny`⟩`Value()` dające wartość typu pierwotnego przechowywaną w obiekcie, np.:

    ```
    ch.charValue()
    bool.booleanValue()
    ```

  – `equals(Object o)` zwracająca wartość `true`, gdy parametr jest różny od `null` i reprezentuje obiekt odpowiedniego typu, z wartością równą wartości przechowywanej w danym obiekcie

  – `valueOf(String s)` zamieniająca łańcuch reprezentujący warość typu pierwotnego na obiekt odpowiedniego typu (z wyjątkiem klasy `Character`), np.:

    ```
    Integer.valueOf("1234")
    Boolean.valueOf("TRUE") // każdy inny łańcuch to false
    ```

- statyczne pola (z wyjątkiem klasy `Boolean`):

  ```
  MIN_VALUE
  MAX_VALUE
  ```

W Javie 5.0 zaimplementowano automatyczne opakowywanie i rozpakowywanie wartości typów podstawowych. Dzięki temu nie trzeba dokonywać jawnych konwersji, co zdecydowanie upraszcza kod, np:

```
Integer x, y;
x = 12;  // automatyczne opakowanie
y = 15;  // automatyczne opakowanie
System.out.println(x+y); // automatyczne rozpakowanie
```

Tablice

Tablica w Javie jest kontenerem elementów z ustaloną liczbą pojedynczych wartości. Wielkość (liczba elementów tablicy) jest ustalana na etapie jej tworzenia. Elementy tablicy są numerowane liczbami całkowitymi nieujemnymi począwszy od 0. Cechy tablic Javy można określić następująco:

- elementy mogą być dowolnego typu — podstawowego lub obiektowego,
- istnieją tylko tablice jednowymiarowe,
- tablice są obiektami, każda tablica posiada pole `length` zawierające liczbę elementów tablicy,
- w odróżnieniu od języka C++, nazwa tablicy nie może być traktowana jak wskaźnik na pierwszy element.

Java rozróżnia etapy: deklarowania zmiennej tablicowej, tworzenia tablicy oraz inicjowania tablicy (który jest opcjonalny), po których zmienna tablicowa może być użyta do przechowywania danych.

Deklarowanie zmiennych tablicowych polega na podaniu typu elementów oraz nazwy tablicy:

```
⟨typElementu⟩ ⟨nazwa⟩[];   // stary model
⟨typElementu⟩[] ⟨nazwa⟩;   // nowy model, zalecany
```

na przykład:

```
char[]   tablicaZnaków;
int[]    tablicaLiczbCałkowitych;
float[]  tabliczLiczbFloat;
Object[] tablicaObiektów;
String[] tablicaŁańcuchów;
```

Po zadeklarowaniu, tablica musi zostać utworzona na jeden z dwu sposobów: za pomocą operatora `new` (może to być połączone z deklaracją):

```
⟨nazwa⟩ = new ⟨typElementu⟩[⟨rozmiar⟩];
// ⟨rozmiar⟩ może być wyrażeniem
// (typu całkowitego o wartości dodatniej)
```

na przykład:

```
[commandchars=\\\{\}]
int[]   tabInt = new int[50];
Float[] tabFl;
char[]  tabZn;
. . .
tabZn  = new char[100];
tabFl  = new Float[2*n+1];
```

lub za pomocą wyrażenia inicjującego (tylko z deklaracją):

⟨typElementu⟩[] ⟨nazwa⟩ = {⟨w_0⟩, ⟨w_1⟩, ..., ⟨w_{n-1}⟩};

które tworzy tablicę o n elementach. Wyrażenie inicjujące może zawierać dowolne wyrażenia typu zgodnego z typem elementów tablicy, na przykład:

```
char[] tabZn = {'A', 'l', 'a', ' ', 'i', ' ', 'A', 's'};
// nie można inicjować łańcuchem znaków
double[] tabInt = {Math.PI, 1.5, Math.exp(1), '5', 7, 2*k+5};
```

W momencie tworzenia tablicy można ją zainicjować na zasadach opisanych powyżej, przy czym ta operacja może być wykonana poza deklaracją zmiennej tablicowej — tworzona jest wtedy nowa tablica z określoną zawartością, na przykład:

```
// inicjowanie w miejscu deklarowania i tworzenia tablicy
char[]   tZn  = new char[]{'A', 'l', ' ', 'i', ' ', 'A', 's'};
double[] tDbl = new double[]{1.5, 2.1, -3.6};
// inicjowanie w momencie tworzenia tablicy poza deklaracją
// boolean[] tabBool;
tabBool = new boolean[]{true, false, false, true};
// int[] tabInt;
tabInt = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Java nie udostępnia tablic wielowymiarowych — ich funkcję pełnią tablice tablic. Syntaktyka tych tablic przedstawia się następująco:

⟨typElementu⟩[][]...[] ⟨nazwa⟩; // zalecany nowy model

na przykład:

```
int [][] tabInt2w;
boolean[][][] tabBool3w;
```

Tworzenie i inicjowanie tablic tablic przebiega podobnie jak poprzednio, z tym, że przy tworzeniu tablic za pomocą operatora `new` przynajmniej pierwszy „wymiar" tablicy musi być określony przy ich tworzeniu:

⟨typElementu⟩[][]...[] ⟨nazwa⟩
= new ⟨typElementu⟩[n_1][n_2]...[n_m];
⟨nazwa⟩ = new ⟨typElementu⟩[n_1][n_2]...[n_m];
// przynajmniej n_1 musi być określone

```
⟨typElementu⟩[][]...[] ⟨nazwa⟩ = ⟨inicjator⟩;
⟨typElementu⟩[][]...[] ⟨nazwa⟩
                        = new ⟨typElementu⟩[][]...[]⟨inicjator⟩;
⟨nazwa⟩ = new ⟨typElementu⟩[][]...[]⟨inicjator⟩;
```

Należy również pamiętać, że są to tablice tablic, stąd ⟨`inicjator`⟩ musi mieć także postać tablicy tablic, na przykład:

```
int[][] tab2w = {{ 1, 2 }, { 3, 4 }, { 5, 6 }};
```

Poniższy przykład podsumowuje podstawowe informacje o tablicach.

```
public class t2w {
  public static void main(String[] args)
    { int[][] tab2w = new int[4][];
      // wypełnij tablice
      tab2w[0]=new int[]{0,1};
      for ( int w = 1; w < tab2w.length; w++ )
        { tab2w[w] = new int[w+2]; // twórz tablice wewnętrzne
          for ( int k = 0; k < tab2w[w].length; k++ )
            tab2w[w][k] = w + k;
        }
      // pisz tablice
      for ( int w = 0; w < tab2w.length; w++ )
        { for ( int k = 0; k < tab2w[w].length; k++ )
            System.out.print( tab2w[w][k] + " " );
          System.out.println();
        }
    }
/********
0 1
1 2 3
2 3 4 5
3 4 5 6 7
********/
```

Wiele operacji na tablicach udostępnia również klasa `Arrays`. Są to statyczne metody (dla każdej istnieje kilka przeciążonych wersji):

`equals(...)` — sprawdzenie identyczności tablic,

`fill(...)` — wypełnienie tablicy określoną wartością,

`sort(...)` — sortowanie,

`binarySearch(...)` — wyszukiwanie binarne,

`asList(...)` — zamiana tablicy na listę.

Na szczególną uwagę zasługują metody: `sort(...)` oraz `binarySearch(...)`. Gdy mają działać na tablicy obiektów, trzeba przygotować metodę porównującą elementy tablicy. W tym celu należy:

1. w klasie definiującej elementy tablicy zaimplementować interfejs `Comparable`,
2. zdefiniować metodę `compareTo( Object o )` opisującą sposób porównywania elementów tablicy.

Poniższy przykład ilustruje opisane operacje.

```
public class CompType implements Comparable
{ private int wart;
  public CompType( int wart_ )
    { wart = wart_;
    }
  public int compareTo(Object o)
    { int arg = ( (CompType)o ).wart;
      if( wart == arg ) return 0;
      if( wart < arg ) return -1;
      return 1;
    }
  public static void print( Object[] tab )
    { for( int i = 0; i < tab.length; i++ )
        System.out.print( tab[i] + " " );
      System.out.println();
    }
  public String toString() { return wart + ""; }
  public static void main(String[] args )
    { CompType[] tab = new CompType[20];
      for( int i = 0; i < tab.length; i++ )
        tab[i] = new CompType((int)( Math.random() * 100 ));
      print(tab);
      Arrays.sort( tab );
      print( tab );
      int szuk = Arrays.binarySearch( tab, tab[3] );
      System.out.println( "Położenie szukanego elementu: "
                                  + tab[3] + " = " + szuk );
    }
}
/*********************************************************
31 74 96 87 92 69 39 76 81 40 34 71 13 95 22 34 80 19 79 94
13 19 22 31 34 34 39 40 69 71 74 76 79 80 81 87 92 94 95 96
Położenie szukanego elementu 31 = 3
*********************************************************/
```

Łańcuchy

W Javie łańcuchy znaków są obiektami. Najważniejszymi klasami umożliwiającymi manipulowanie łańcuchami znaków są `String` oraz `StringBuffer` (a także `StringBuilder`).

Obiekty klasy `String` reprezentują łańcuchy niemodyfikowalne. Oznacza to, że każda próba zmiany wartości takiego łańcucha powoduje utworzenie nowej instancji (danej) zawierającej zmodyfikowaną

wartość, co jest pewną wadą. Z drugiej strony, dla łańcuchów tej klasy przeciążono operator „+”, dzięki czemu wygodnie można wykonywać operacje składania takich łańcuchów (dla łańcuchów pozostałych klas taką operację należy wykonać wywołując odpowiednie metody).

Tworzenie łańcuchów klasy `String` odbywa się na kilka sposobów:

1. za pomocą literałów,
2. jako wynik składania literałów: ł1 + ł2,
3. jako wynik wykonania konstruktora (klasa `String` posiada 13 konstruktorów), np.:
   - `String()` — tworzy łańcuch pusty ("”),
   - `String( String łańc )` — tworzy łańcuch jako kopię argumentu,
   - `String( char[] tablica )` — tworzy łańcuch na podstawie tablicy znaków.

Poniżej pokazano przykłady tworzenia łańcuchów:

```
String s = "Ala ma kota";
String s = "Ala" + "  ma" + "  kota";
String s = String();
String s = String( "Ala ma kota" );
char[] tabZn = { 'A', 'l', 'a', ' ', 'm', 'a',
                 ' ', 'k', 'o', 't', 'a' };
String łańcuch = new String( tabZn );
```

Przykładowe operacje, jakie udostępnia klasa `String` to:

- `char charAt( int index )` — znak o określonym indeksie (liczone od 0),
- `int compareTo( String drugi )` — porównanie dwóch łańcuchów:

$$\text{wynik} = \begin{cases} < 0 & \text{— łańcuch „mniejszy” niż argument} \\ = 0 & \text{— łańcuchy równe (identyczne)} \\ > 0 & \text{— łańcuch „większy” niż argument} \end{cases}$$

- `concat( String str )` — połączenie łańcuchów
- `endsWith( String suffix )` — czy argument jest początkiem łańcucha
- `equals( Object obiekt )` — czy łańcuchy równe (identyczne)
- `indexOf( String str, int fromIndex )` — numer znaku rozpoczynajacego pierwszy podłańcuch
- `lastIndexOf(String str, int fromIndex)` — numer znaku rozpoczynajacego ostatni podłańcuch

- `length()` — długość łańcucha (metoda)
- `regionMatches( int offset, String str, int offsetStr, int dł )` — czy fragmnety łańcuchów równe
- `replace( char oldChar, char newChar )` — zamiana znaków
- `startsWith( String prefix, int toffset )` — czy łańcuch zaczyna się od podłańcucha
- `substring( int beginIndex, int endIndex )` — `endIndex` liczony wyłącznie
- `toCharArray()` — zamiana łańcucha na tablicę znaków
- `toLowerCase()` — zamiana na małe litery (z polskimi literami)
- `toUpperCase()` — zamiana na duże litery (z polskimi literami)
- `trim()` — wycięcie skrajnych spacji
- `valueOf(`⟨typ⟩` arg )` — zamiana wartości typu ⟨typ⟩ na łańcuch (z kolei podklasy klasy `Number` (`Byte`, `Integer`, `Double`, `Float`, `Long` i `Short`) mają metody `valueOf`, które konwertują łańcuch znaków na wartość określonego typu).

Należy zwrócić uwagę, że literały łańcuchowe też są obiektami, co umożliwia wywoływanie na ich rzecz metod, na przykład:

```
"hamburger".substring(0, 7)  // "hamburg"
"łąka".toUpperCase()         // "ŁĄKA"
```

Oczywiście, ponieważ obiekty typu `String` reprezentują łańcuchy niemodyfikowalne, wykonanie odpowiednich metod nie zmienia wartości łańcucha, na rzecz którego matoda została wykonana. Zmieniony łańcuch pojawia się jako wynik wywołanej metody. Działanie wybranych metod prezentuje poniższy przykład:

```
public class Lancuch {
public static void main(String[] args )
{ String str = new String();
    int wyn, startPos;
    char zn = 'a';

    str = "  Ala ma kota  ";
    wyn = (startPos = str.indexOf( zn )) < 0 ? -1
          : str.lastIndexOf( zn ) - startPos - 1;
    System.out.println( "wyn = "+wyn );
    // wyn = 7
    System.out.println( '"'+str+'"' );
    //  "  Ala ma kota  "
    System.out.println( '"'+str.replace('a','*')+'"' );
    //  "  Al* m* kot*  "
    System.out.println( '"'+str.toLowerCase()+'"' );
    //  "  ala ma kota  "
```

```
    System.out.println( '"'+str.toUpperCase()+'"' );
    //  "  ALA MA KOTA  "
    System.out.println( '"'+str.trim()+'"' );
    //  "Ala ma kota"
    System.out.println( str.compareTo("  ALA")); // 32
    System.out.println( str.compareTo("  ala")); // -32
    System.out.println( str.compareTo("  Ala ma kota  ")); // 0
    System.out.println( str.equals("  Ala ma kota  "));
    // true
    System.out.println( str.equals("  ALA MA KOTA  "));
    // false
    System.out.println( str.equalsIgnoreCase("  ALA MA KOTA  ") );
    // true
    System.out.println( str.equalsIgnoreCase(str.toUpperCase()) );
    // true
  }
}
```

W odróżnieniu od klasy `String`, która reprezentuje łańcuchy niemodyfikowalne, przetwarzanie łańcuchów bez każdorazowego tworzenia ich nowych instancji może odbywać sie stosując klasę `StringBuffer` (lub `StringBuilder`, która ma identyczne właściwości, co `StringBuffer`, lecz nie powinna być stosowana w przypadku wielowątkowości, ze względu na brak synchronizacji metod).

Obiekty typu `StringBuffer` są implementowane jako zmiennej wielkości tablice mogące zawierać sekwencje znaków. W związku z tym każdy taki obiekt oprócz aktualnie przechowywanego łańcucha ma swoją pojemność (w odróżnieniu od łańcuchów typu `String`) określającą maksymalną długość łańcucha, jaki może być w nim umieszczony (może ona być odczytana za pomocą metody `capacity()`). Ta pojemność jest zawsze większa lub równa długości przechowywanego łańcucha i zostaje automatycznie powiększana przy wydłużaniu pamiętanego łańcucha (np. przy wykonywaniu metody `append(...)` lub `insert(...)`).

Klasa `StringBuffer` ma trzy konstruktory:

- `StringBuffer()` — tworzy pusty łańcuch o pojemności 16 znaków,
- `StringBuffer( int dł )` — tworzy pusty łańcuch o pojemności `dł` znaków,
- `StringBuffer( String łańc )` — tworzy łańcuch zainicjowany kopią podanego łańcucha,

niektóre metody identyczne jak klasa `String` (np. `charAt(...)`, `getChars(...)`, `indexOf(...)`, `lastIndexOf(...)`, `length()`, `substring(...)`) oraz zbiór charakterystycznych metod zmieniających wartość obiektu, na rzecz którego są wywoływane, na przykład:

- `toString()` — konwersja na łańcuch typu `String`,

- `setLength( int nowaDł )` — zmiana długości łańcucha (ewentualne dodanie na koniec łańcucha znaków '\u0000',
- `setCharAt( int index, char zn )` — zamiana znaku na pozycji `index`,
- `append(...)` — dołączenie łańcucha,
- `insert(...)` — wstawienie łańcucha ,
- `delete( int pocz, int kon )` — usunięcie fragmentu łańcucha,
- `replace( int pocz, int kon, String łańc )` — zamiana fragmentu na łańcuch `łańc`,
- `reverse()` — odwrócenie łańcucha.

Ważne, że metody `append(...)` oraz `insert(...)` są przeciążone, tak aby akceptować dane dowolnego typu. Metody te najpierw konwertują argument na łańcuch znaków, a następnie wykonują odpowiednią akcję.

Oczywiście obiekty klasy `StringBuffer` zawsze można konwertować na typ `String`, wywołać dla takiego obiektu potrzebną metodę klasy `String` (której brak w klasie `StringBuffer`), a następnie powrócić do klasy `StringBuffer` (lub odwrotnie), np.:

```
String łańc = "Ala ma kota";
łańc = new StringBuffer( łańc ).reverse().toString();

StringBuffer sb = new StringBuffer("Ala ma kota");
sb.replace(0, sb.length(), sb.toString().toUpperCase());
```

Właściwości klas operujących na łańcuchach można przedstawić na przykładzie odwracania łańcuchów. Pierwszy z nich operuje na łańcuchach `String` przez co generuje wiele łańcuchów „zaśmiecających" pamięć, ale za to kod nie jest zbyt obszerny:

```
public class OdwróćŁańcuchStr
  { public static void main(String[] args)
      { String łańcuch = "może jutro ta dama da tortu jeżom";
        String odwrŁańcuch = new String();
        // odwrócenie łańcucha
        for (int j = 0; j < łańcuch.length(); j++)
            odwrŁańcuch = łańcuch.charAt( j ) + odwrŁańcuch;
        System.out.println( odwrŁańcuch );
      }
  }
```

Przykład ten można nieco rozbudować wykorzystując do odwracania łańcucha tablicę znaków, przez co zmniejsza się obciążenie pamięci kosztem wydłużenia kodu programu:

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

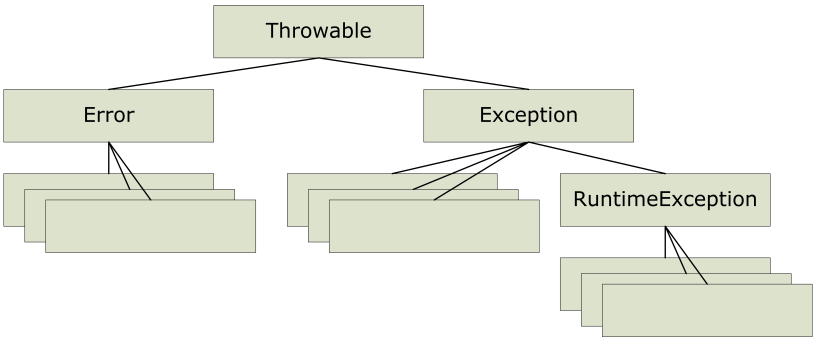
```
    { String łańcuch = "może jutro ta dama da tortu jeżom";
      char[] tabZn = łańcuch.toCharArray();
      int dł = łańcuch.length();
      // odwrócenie tablicy
      for (int j = 0; j < dł; j++)
          tabZn[j] = tabZn[dł - 1 - j];
      String odwrŁańcuch =  new String( tabZn );
      System.out.println( odwrŁańcuch );
    }
}
```

Po zastąpieniu klasy `String` klasą `StringBuffer` można zastosować metodę `reverse()`, co zarówno skróci kod, jak i zmniejszy wielkość zużytej pamięci:

```
public class OdwróćŁańcuchSB
  { public static void main(String[] args)
      { StringBuilder łańcuch =
          new StringBuilder("może jutro ta dama da tortu jeżom");
        łańcuch.reverse();  // odwrócenie łańcucha
        System.out.println(łańcuch);
      }
  }
```

Obsługa wyjątków

Jak w większości nowoczesnych języków programowania, w Javie również wyjątki są obsługiwane z zastosowaniem podejścia obiektowego. Jak zwykle w momencie wystąpienia błędu w czasie wykonywania metody, zostaje utworzony obiekt zawierający informacje o błędzie związane z jego typem oraz stanem programu w momencie jego wystapienia. Rodzaj wyjątku jest reprezentowany przez typ klasy, której obiekt zostanie utworzony. Wszystkie te klasy dziedziczą po klasie `Throwable` (rys. 30).



Rysunek 30: Hierarchia klas obsługi wyjątków

W Javie rozróżnia się trzy rodzaje wyjątków:

1. kontrolowane (zwane także obsługiwalnymi) — błędy, które dobrze napisana aplikacja powinna przewidywać i na nie reagować (np. błędna nazwa pliku podana przez użytkownika),

2. błędy z przyczyny zewnętrznej w stosunku do aplikacji (np. błąd sprzętu), reprezentowane przez klasę `Error` i potomne,
3. błędy wykonania programu powstałe wewnątrz aplikacji z powodu błędu programisty (np. błędy logiczne, których nie wychwycił kompilator), reprezentowane przez klasę `RuntimeException` i potomne.

Wyjątki wynikające z błędów zewnętrznych i błędów wykonania są nazywane niekontrolowanymi. Przykładami wyjątków kontrolowanych mogą być:

- FileNotFoundException (próba dostępu do nieistniejącego pliku),
- UnknownHostException (błędny adres dostępu do zasobu Internetu),
- SQLException (błąd dostępu do zasobów bazy danych),

a wyjątków niekontrolowanych:

- podklasy klasy `Error`:
  - `AWTError` (błąd biblioteki AWT),
  - `LinkageError` (np. niepoprawne zależności pomiędzy klasami),
  - `ThreadDeath` (próba zatrzymania wątku, który już nie działa),
  - `VirtualMachineError` (błąd działania JVM, np. brak zasobów),
- podklasy klasy `RuntimeException`:
  - `ArithmeticException` (problem z wykonaniem operacji arytmetycznej, np. próba dzielenia przez zero),
  - `ArrayIndexOutOfBoundsException` (odwołanie do nieistniejącego elementu tablicy),
  - `OutOfMemoryError` (brak pamięci),
  - `StackOverflowError` (przepełnienie pamięci stosu).

Konstrukcja odpowiadająca za obsługę wyjątków ma następującą budowę:

```
try {
... // kod mogący wygenerować wyjątek
    }
catch( ⟨typWyjątku1⟩ ⟨zm1⟩ ) { ⟨czynnościObsługi1⟩ }
catch( ⟨typWyjątku2⟩ ⟨zm2⟩ ) { ⟨czynnościObsługi2⟩ }
...
catch( ⟨typWyjątkun⟩ ⟨zmn⟩ ) { ⟨czynnościObsługin⟩ }
finally { ⟨czynnościKończące⟩ }
```

gdzie ⟨typWyjątku$_i$⟩ jest nazwą klasy reprezentującej wyjątek przechwytywany przez ten element `catch` (przechwytuje też wyjątki klas potomnych), co powoduje wykonanie ⟨czynnościObsługi$_i$⟩, natomiast część `finally` jest wykonywana zawsze, niezależnie od tego, czy wyjątek wystąpił, czy też nie. Zmienne ⟨`zm`$_i$⟩ reprezentują obiekty odpowiednich klas, które jako podklasy klasy `Throwable` udostępniają między innymi metody:

- `getMessage()` — tekst komunikatu o błędzie,
- `prinStackTrace()` — wyprowadza nazwę wyjątku oraz stos metod aktywnych w momencie wystąpienia wyjątku,
- `toString()` — opis wyjątku:
  - nazwa klasy wyjątku,
  - ": " (dwukropek ze spacją),
  - wynik wywołania metody `getMessage()` dla tego obiektu.

Poniższy przykład pokazuje zasadę działania konstrukcji `try-catch-finally`:

```
public class Finally
  { static int licznik = 0;
    public static void main( String[] args )
      { int x;
        while( licznik < 3 )
          try
            { if( licznik++ == 1 ) x = 1 / 0;
              if( licznik == 3 ) System.out.print( "znowu " );
              System.out.println( "bez wyjątku" );
            }
          catch( Exception e )
            { System.out.println( "wyjątek !!! - "+e.getMessage());
              e.printStackTrace();
            }
          finally
            { System.out.println( "jestem w \"Finally\"" );
            }
       }
  }
/******** wynik działania: **************************
bez wyjątku
jestem w "Finally"
wyjątek !!! - java.lang.ArithmeticException: / by zero
java.lang.ArithmeticException: / by zero
        at finall.Finally.main(Finally.java:9)
jestem w "Finally"
znowu bez wyjątku
jestem w "Finally"
*****************************************************/
```

Jeżeli wyjątek nie jest obsłużony jawnie (w kontekście, w którym wystąpił lub w kontekstach nadrzędnych), jest przechwycony przez procedurę domyślną, która wyświetla komunikat błędu, stan stosu wywołań funkcji i powoduje zakończenie aplikacji.

Java wymaga jawnego określenia wyjątków, jakie może zgłosić dana metoda. Polega to na tym, że jeżeli metoda zgłasza wyjątek, którego sama nie obsługuje, to musi opisać takie zachowanie w swoim nagłówku. Służy do tego lista typów wyjątków po słowie

kluczowym `throws` umieszczona po deklaracji (nagłówku) metody. Należy rozpartywać jedynie wyjątki kontrolowane (czyli nie uwzględnia się wyjątków typu `Error` i `RuntimeException` wraz z ich podklasami). Ogólna postać definicji metody zgłaszającej wyjątki jest następująca:

```
⟨typWyniku⟩ ⟨nazwaMetody⟩(⟨listaParametrów⟩)
        throws ⟨listaWyjątków⟩
     {
         // treść metody
     }
```

gdzie ⟨`listaWyjątków`⟩ jest ciągiem nazw klas wyjątków rozdzielonych przecinkami. Każda taka metoda może zostać wywołana jedynie w zakresie `try-catch`.

W Javie można także samemu zgłaszać wyjątki. Służy do tego instrukcja `throw`, która może wznowić wyjątek właśnie obsługiwany w części `catch` (nie zmienia się wtedy rodzaj wyjątku) lub zgłosić wyjątek dowolnego typu, gdy słowu `throw` towarzyszy obiekt odpowiedniej klasy — poklasy klasy `Throwable` — „standardowej" lub stworzonej przez użytkownika, np.:

```
throw;          // wznowienie wyjątku
throw new EmptyStackException();
throw new MojaKlasaWyjątku("Nieznany błąd");
```

Poniższy przykład pokazuje zgłaszanie oraz obsługę własnych wyjątków (w przykładzie jest obsługiwany wyjątek dzielenia przez zero, który należy do grupy wyjątków niekontrolowanych, jednak chodzi tutaj jedynie o pokazanie mechanizmów obsługi wyjątków niezależnie od ich typu):

```
1  public class Wyjątki
2  { float Średnia1( int suma, int ile )
3    { if (ile !=0 ) return suma / ile;
4        else return 0;
5    }
6
7    float Średnia2( int suma, int ile )
8    { try
9      { return suma / ile;
10     }
11     catch( ArithmeticException e )
12     { System.out.println("Wyjątek: "+e.toString());
13       return 0;
14     }
15   }
16
17   float Średnia3( int suma, int ile ) throws Exception
18   { try
19     { return suma / ile;
20     }
```

```
21     catch( ArithmeticException e )
22     { System.out.println("Wyjątek: "+e.getMessage());
23       throw new Exception("Dzielenie przez zero!!!");
24     }
25   }
26
27   float Średnia4( int suma, int ile ) throws MójWyjątek
28   { try
29     { return suma / ile;
30     }
31     catch( ArithmeticException e )
32     { System.out.println("Wyjątek: "+e.getMessage());
33       throw new MójWyjątek("Dzielenie przez zero!!!");
34     }
35   }
36
37   public static void main(String[] args)
38   { Wyjątki w = new Wyjątki();
39
40     System.out.println("Średnia1 = " + w.Średnia1( 25, 10));
41 // Średnia1 = 2.0
42     System.out.println("Średnia1 = " + w.Średnia1( 25, 0));
43 // Średnia1 = 0.0
44
45     System.out.println("Średnia2 = " + w.Średnia2( 25, 10));
46 // Średnia2 = 2.0
47     System.out.println("Średnia2 = " + w.Średnia2( 25, 0));
48 // Wyjątek: java.lang.ArithmeticException: / by zero
49 // Średnia2 = 0.0
50
51     try
52     { System.out.println("Średnia3 = " + w.Średnia3( 25, 10));
53 // Średnia3 = 2.0
54       System.out.println("Średnia3 = " + w.Średnia3(25, 0));
55     }
56     catch( Exception e )
57     { System.out.println("Wyjątek: "+e.getMessage());
58     }
59 // Średnia3 = 2.0
60 // Wyjątek: / by zero
61 // Wyjątek: Dzielenie przez zero!!!
62
63     try
64     { System.out.println("Średnia4 = " + w.Średnia4( 25, 10));
65 // Średnia4 = 2.0
66       System.out.println("Średnia4 = " + w.Średnia4(25, 0));
67     }
68     catch( MójWyjątek e )
69     { System.out.println("Wyjątek: "+e.toString());
```

```
70     }
71 //********** z super( msg ) **********************
72 // Średnia4 = 2.0
73 // Wyjątek: / by zero
74 // wyjątki.MójWyjątek: Dzielenie przez zero!!!
75 //********** bez super( msg ) **********************
76 // Średnia4 = 2.0
77 // Wyjątek: / by zero
78 // wyjątki.MójWyjątek
79   }
80 }
81
82 class MójWyjątek extends Exception
83 { MójWyjątek( String msg )
84   { super( msg ); // metoda super(...} może być wywołana lub nie
85   }
86 }
```

W prezentowanym przykładzie funkcja (metoda) `Średnia1` realizuje obsługę błędu dzielenia przez zero w sposób bezpośredni (bez wyjątków) — przewidując wartość zerową mianownika. Funkcja `Średnia2` obsługuje wyjątek `ArithmeticException` wypisując pełną informację o sytuacji wyjątkowej (metoda `toString`). Z kolei funkcja `Średnia3` oprócz wypisania komunikatu związanego z wyjątkiem (metoda `getMessage`) generuje wyjątek typu `Exception` z komunikatem "Dzielenie przez zero!!!" (linia 23). Ta sytuacja wymaga zadeklarowania tego wyjątku w nagłówku funkcji (linia 17) oraz wywołania funkcji `Średnia3` w obrębie konstrukcji `try-catch` (linia 57), gdzie tym razem zostaje wyprowadzony komunikat będący parametrem konstruktora wywoływanego w linii 23. Wreszcie w funkcji `Średnia4`, podobnie jak w przypadku funkcji `Średnia3`, następuje obsługa wyjątku poprzez wypisanie oryginalnego komunikatu, ale tym razem zostaje wygenerowany wyjątek typu `MójWyjątek` (linia 33), który z kolei jest przechwycony w linii 68. W deklaracji funkcji `Średnia4` również został zadeklarowany wyjątek — tym razem typu `MójWyjątek`. Należy tutaj zauważyć różnicę w działaniu metody `toString` (linia 69) gdy konstruktor klasy `MójWyjątek` wywołuje jawnie konstruktor klasy `Exception` z parametrem `msg` zawierającym komunikat określony w linii 33 (za pomocą funkcji `super( msg )`) od sytuacji, gdy funkcja `super` nie zostaje wywołana. W pierwszym przypadku metoda `toString` wyprowadzi zarówno opis wyjątku, jak i komunikat o błędzie (linia 74), podczas gdy w drugim — jedynie opis wyjątku (linia 78), gdyż bez wywołania metody `super` zostaje automatycznie wywołany konstruktor bezparametrowy klasy `Exception`, przez co w utworzonym obiekcie tej klasy nie zostaje zapamiętany komunikat będący parametrem konstruktora klasy `MójWyjątek` wywołanego w linii 33.

Zadania

Zad. 1. Zdefiniować klasę opisującą funkcję kwadratową o postaci:

$$y = ax^2 + bx + c$$

Współczynniki a, b, c powinny być prywatne. Zdefiniować konstruktor i publiczne funkcje składowe:

- konstruktor nadający wartości współczynnikom,
- funkcję obliczającą wartość y dla podanego x,
- funkcję wyznaczającą liczbę pierwiastków równania kwadratowego:

$$ax^2 + bx + c$$

Zad. 2. Napisać klasę `Samochód` zawierającą pola `marka` (łańcuch znaków) i `vMax` (liczba całkowita) z konstruktorami: inicjującym dane na wartości z parametrów i kopiującym dane oraz metodą wypisującą dane na ekran. Zdefiniować klasy pochodne: `Autobus`, dodającą pole `lMiejsc` (liczba całkowita) oraz `Ciężarówka`, dodającą pole `nośność` (liczba całkowita). Klasy pochodne powinny zawierać odpowiednie konstruktory oraz metody wypisujące dane na ekran.

Zad. 3. Zdefiniować klasę obsługującą posortowaną kolejkę elementów typu `int`. Zdefiniować następujące funkcje:

- konstruktor zakładający pustą kolejkę o określonej pojemności,
- konstruktor kopiujący kolejkę,
- funkcje `Wstaw( element )`, `Usuń( element )` oraz `Pobierz( element )`.

Zad. 4. Zdefiniować klasę obsługującą stos elementów typu `double`. Zdefiniować następujące funkcje:

- konstruktor zakładający pusty stos o określonej pojemności,
- konstruktor kopiujący stos,
- funkcje `Push( element )` oraz `Pop()`.

Zad. 5. Zaimplementować klasę `Zbiór` zapamiętującą elementy na bajtach. Klasa powinna zawierać konstruktor(y) oraz funkcje: `Suma`, `Iloczyn`, `Różnica`, `CzyscZbior` i `CzyWZbiorze`.

Zad. 6. Zdefiniować klasę opisu daty `TDate`, umożliwiającą przechowywanie daty w polach `dzień`, `miesiąc`, `rok`. Klasa ma posiadać odpowiednie konstruktory oraz funkcję składową do konwersji daty:

```
int Wpisz( int dd, int mm, int rr )
```

zwracającą 0, gdy dane są poprawne i 1, gdy nie, oraz funkcję do wyprowadzania daty na wyjście w formacie „dd-mm-rrrr”.

Zad. 7. Uzupełnić kod następującej metody, aby była poprawna:

```
public static void Metoda(File file)
  { RandomAccessFile input = null;
    String wiersz = null;
    try {
        input = new RandomAccessFile( file, "r");
        while (( wiersz = input.readLine()) != null )
          System.out.println( wiersz );

        return;
        }
    finally
      { if (input != null) input.close();
      }
  }
```

Zad. 8. Zdefiniować klasę opisu czasu `TTime`, umożliwiającą przechowywanie czasu w polach `godzina`, `minuta`, `sekunda`. Klasa ma posiadać konstruktor domyślny, wpisujący czas do pól, a także metody do ustawiania i pobierania czasu oraz wyprowadzania czasu na wyjście w formacie „godzina:minuta:sekunda” (każdy element złożony z dwóch cyfr).

Zad. 9. Zdefiniować klasy `Pojazd` i `Samochód` w taki sposób, aby klasa `Pojazd` była klasą bazową dla klasy `Samochód`. Klasa `Pojazd` ma zawierać prywatne pole `nrRejestracyjny` (7 znaków) oraz funkcje ustawiania/pobierania tego numeru. Klasa `Samochód` powinna mieć dodatkowe pole prywatne `marka` (20 znaków) oraz funkcje ustawiania/pobierania wartości tego pola.

Zad. 10. Poprawić błędne instrukcje, zakładając, że klasy `KlG` oraz `KlC` implementują interfejs `IntU`:

```
KlG  p1 = new KlC();
KlG  p2 = new IntU();
IntU p3 = new IntU();
IntU p4 = new KlC();
KlG  p5 = new IntU()
IntU p6 = new p5;
IntU p7;
p7 = new KlC();
```

Haskell

Ogólne informacje o języku

Haskell[27] jest językiem funkcyjnym, charakteryzującym się „silnym typowaniem", „leniwym wartościowaniem" oraz możliwością stosowania paradygmatu programowania obiektowego. Ostatnia z wymienionych cech zostanie omówiona oddzielnie, w podrozdziale dotyczącym programowania obiektowego. Głównym założeniem języków funkcyjnych jest to, że program składa się z definicji funkcji i operatorów oraz wyrażenia złożonego z tych funkcji, operatorów oraz stałych, którego obliczenie daje wynik — dane wyjściowe. Taka koncepcja stoi w opozycji do programowania imperatywnego (C++, Pascal i inne języki), w którym program składa się z ciągu instrukcji; czasem rozgałęzianych, zapętlanych i łączonych w bloki nazywane podprogramami. Silne typowanie oznacza, że niemożliwa jest na przykład przypadkowa konwersja `Double` do `Int`. Prowadzi to do zmniejszenia liczby powstałych błędów. Każda funkcja i każdy argument ma określony typ. Jeśli typu nie określimy jawnie, to zostanie to uczynione automatycznie na etapie kompilacji. Leniwe wartościowanie oznacza, że wartość żadnego wyrażenia nie jest wyznaczana dopóki nie jest wymagana. Można, na przykład, zdefiniować nieskończenie długą listę liczb Fibonacciego (składnia listy zastosowana w przykładzie wymaga użycia parametru kompilatora `-fglasgow-exts`) tworzoną w niekończących się wywołaniach rekurencyjnych:

[27] Wszystkie przykłady umieszczone w tym rozdziale zostały przetestowane w systemie The Glasgow Haskell Compiler (wersja 6.10.4), który wraz z dokumentacją jest dostępny na internetowej stronie `www.haskell.org/ghc/`

```
fib = 0 : 1 : [x+y | x <- fib | y <- tail fib]
```

Wyznaczone zostaną tylko te elementy listy, które są aktualnie potrzebne:

```
> take 13 fib
[0,1,1,2,3,5,8,13,21,34,55,89,144]
```

Leniwe wartościowanie gwarantuje, że nic nie zostanie policzone niepotrzebnie.

Warto także pamiętać o pozostałych cechach wyróżniających Haskella na tle innych języków:

- jest językiem czystym (ang. *purely functional language*) — nie ma efektów ubocznych — za każdym razem gdy wywołamy funkcję z takimi samymi parametrami, zwróci ona taką samą wartość,

gdyż nie mamy do czynienia z ingerencją z zewnątrz (jak np. pobranie danej od użytkownika);

- jest zwięzły — programy są krótsze, dzięki czemu łatwiej jest analizować poszczególne funkcje i lokalizować błędy;
- jest językiem bardzo wysokiego poziomu — programy napisane w Haskellu często są podobne do opisu algorytmu;
- zarządza pamięcią — programista zajmuje się jedynie implementacją algorytmu, a nie zarządzaniem pamięcią;
- funkcje mogą być polimorficzne i istnieje możliwość przeciążania operatorów;
- brak pętli; efekt pętli uzyskujemy w programowaniu funkcyjnym przez rekurencję;
- wcięcia są elementem syntaktyki języka.

Z systemu GHC (Glasgow Haskell Compiler) możemy korzystać na trzy sposoby: interaktywnie (polecenie `ghci`), wykonując kod zawarty w pliku `Nazwa.hs` (polecenie `runghc Nazwa`) lub kompilując program (`ghc -make Nazwa`). Po wydaniu polecenia `ghci`:

```
$ ghci
GHCi, version 6.10.4: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

system interaktywny jest gotowy do wartościowania wyrażeń. Załadowanie definicji funkcji z pliku tekstowego odbywa się za pomocą komendy `:load`. Listę dostępnych komend z krótkim opisem ich znaczenia można uzyskać po wydaniu polecenia `:?`.

Wbudowane typy i struktury danych

Oprócz wbudowanych typów danych wymienionych i scharakteryzowanych w tabeli 3 warto pamiętać o typie `Complex`, dostępnym po załadowaniu biblioteki `Complex`, który umożliwia wykonywanie obliczeń na liczbach zespolonych. Na przykład, aby obliczyć $\sqrt{8+6i}$, należy napisać:

```
> import Complex
> sqrt (8:+6)
3.0 :+ 1.0
```

W Haskellu korzystamy ze standardowych operatorów, których znaczenie jest takie samo jak w większości języków programowania. Dużą niespodzianką jest możliwość stosowania operatorów tak, jakby były one funkcjami oraz funkcji tak, jakby były one operatorami. W tym celu operator bierzemy w nawias, a funkcję w odwrotne apostrofy. Na przykład:

Nazwa	Opis	Przykład użycia
`Bool`	typ boolowski	`True /= False`
`Int`	liczba całkowita	`2 + 2`
`Integer`	liczba całkowita o nieograniczonym zakresie	`10^123`
`Float`	liczba zmiennoprzecinkowa	`2.123`
`Double`	liczba zmiennoprzecinkowa podwójnej precyzji	`pi`
`Char`	pojedynczy znak	`'a'`
`String`	łańcuch znaków	`"abc ew"`
`[a]`	lista elementów typu `a`	`[1, 2, 3]`
`(a,b)`	krotka, czyli niezmienna lista	`(1, "Ala")`

Tabela 3: Wbudowane typy i struktury danych

```
> (+) 2 3
5
> 5 `div` 4
1
```

Wywołania funkcji zapisujemy w notacji prefiksowej — nazwa funkcji, a następnie argumenty oddzielone spacjami:

```
> mod 7 3
1
> logBase 2 256
8.0
```

Należy przy tym pamiętać, że połączenie funkcji z jej argumentami ma najwyższy priorytet, dlatego wyrażenie `even 1+2` jest niepoprawne składniowo. Aby sprawdzić, czy wynik wyrażenia `1+2` jest parzysty, należy użyć nawiasów:

```
> even (1+2)
False
```

Listy konstruowane są za pomocą nawiasów kwadratowych, a także za pomocą operatora `(:)` zwanego konstruktorem listy. Operator ten jest dwuargumentowy: pierwszym argumentem jest element będący „głową" listy, a drugim — lista będąca „ogonem" tworzonej listy. Na przykład:

```
> 1.6:[3.0,4.2]
[1.6,3.0,4.2]
```

Elementy listy indeksowane są kolejnymi liczbami całkowitymi rozpoczynając od zera. Operatorem zwracającym n-ty element listy jest `!!`, na przykład:

```
> ['a', 'b', 'r'] !! 1
'b'
```

Konkatenację dwóch list realizuje się za pomocą operatora `(++)`, na przykład:

```
> ["Ala", " "] ++ ["ma"]
["Ala"," ","ma"]
```

Biorąc pod uwagę liczbę sposobów definiowania list oraz liczbę standardowych funkcji operujących na listach, Haskell nie ustępuje żadnemu innemu językowi. Na potwierdzenie tej tezy zdefiniujemy w różny sposób listę pięciu pierwszych liczb nieparzystych:

```
> [1,3,5,7,9]
[1,3,5,7,9]
> 1:3:5:7:9:[]
[1,3,5,7,9]
> [1,3..9]
[1,3,5,7,9]
> [2*k - 1 | k <- [1..5]]
[1,3,5,7,9]
> [k | k <- [1..10], odd k]
[1,3,5,7,9]
```

Do najczęściej wykonywanych operacji na listach należą: zwracanie pierwszego elementu (tzw. głowa listy),

```
> head [1,3,5,7,9]
1
```

zwracanie wszystkich elementów oprócz pierwszego (tzw. ogon listy),

```
> tail [1,3,5,7,9]
[3,5,7,9]
```

zwracanie wszystkich elementów oprócz ostatniego,

```
> init [1,3,5,7,9]
[1,3,5,7]
```

zwracanie ostatniego elementu,

```
> last [1,3,5,7,9]
9
```

zwracanie *n* początkowych elementów,

```
> take 3 [1,3,5,7,9]
[1,3,5]
```

pomijanie *n* początkowych elementów,

```
> drop 2 [1,3,5,7,9]
[5,7,9]
```

sprawdzanie przynależności,

```
> elem 8 [1,3,5,7,9]
False
```

zwracanie długości listy,

```
> length [1,3,5,7,9]
5
```

sumowanie elementów (tylko dla listy liczbowej),

```
> sum [1,3,5,7,9]
25
```

odwracanie porządku,

```
> reverse [1,3,5,7,9]
[9,7,5,3,1]
```

zastosowanie funkcji do każdego elementu i zwracanie listy wyników,

```
> map (>6) [1,3,5,7,9]
[False,False,False,True,True]
```

zwracanie listy jedynie z tymi elementami, które spełniają zadany warunek (a dokładniej — dla których przekazana funkcja jednoargumentowa zwraca wartość logiczną `True`),

```
> filter (\x -> rem x 3 == 0) [1,3,5,7,9]
[3,9]
```

łączenie dwóch list w listę par elementów,

```
> zip [1,3,5,7,9] ['a'..'e']
[(1,'a'),(3,'b'),(5,'c'),(7,'d'),(9,'e')]
```

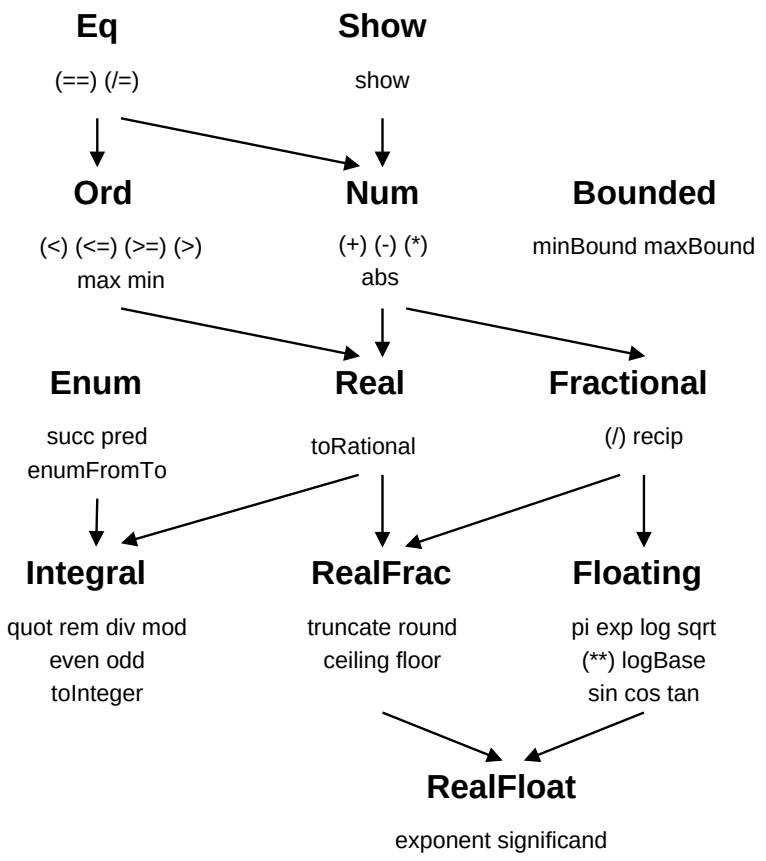
W ostatnim przykładzie elementami listy są krotki typu `(Int, Char)`. Jak łatwo zauważyć typy elementów krotki — w przeciwieństwie do elementów listy — mogą się różnić. Jednak długość krotki jest stała. Bardzo użytecznymi funkcjami działającymi dla krotek dwuelementowych są funkcja `fst` oraz funkcja `snd` zwracające, odpowiednio, pierwszy i drugi element krotki.

Łańcuchy, czyli dane typu `String`, są reprezentowane w Haskellu jako listy znaków. Zatem typ łańcucha jest równoważny typowi `[Char]`. Konsekwencją tego faktu jest możliwość stosowania wszystkich funkcji listowych do łańcuchów.

Wbudowane klasy typów

W Haskellu typy podstawowe pogrupowane są w klasy typów, które spełniają rolę klas abstrakcyjnych. Klasy typów są bardzo podobne do interfejsów znanych z języka Java. Programista ma także możliwość definiowania własnych klas typów i wyprowadzania z nich typów podstawowych oraz typów zdefiniowanych przez siebie. Temat ten zostanie szczegółowo omówiony w jednym z kolejnych podrozdziałów. Na rys. 31 przedstawiono większość klas typów wraz z wybranymi funkcjami, które muszą być zdefiniowane dla typów należących do danej klasy.

Dzięki zastosowaniu klas typów możliwe jest definiowanie funkcji polimorficznych. Na przykład łatwo sprawdzić, że funkcja `sum`:



Rysunek 31: Klasy typów oraz ich wybrane operatory i funkcje; strzałki ilustrują relację dziedziczenia

```
> :type sum
sum :: (Num a) => [a] -> a
```

jest zadeklarowana jako funkcja działająca na listach zawierających dane numeryczne. Ponieważ do klasy `Num` należą liczby całkowite i zmiennoprzecinkowe, funkcję `sum` można stosować do danych kilku typów numerycznych (`Int`, `Integer`, `Float`, `Double`, `Complex`). W tabeli 4 przedstawiono klasy typów oraz typy podstawowe, które są ich egzemplarzami, jak również klasy typów, z których typy wyprowadzono automatycznie (więcej o wyprowadzaniu znajduje się w jednym z kolejnych podrozdziałów).

Tabela 4: Wbudowane klasy i ich typy

Typ	Za pomocą `instance`	Za pomocą `deriving`
`Bool`		`Eq Ord Enum Bounded`
`Int`	`Integral Bounded`	
`Integer`	`Integral`	
`Float`	`RealFloat`	
`Double`	`RealFloat`	
`Char`	`Eq Ord Enum`	
`[a]`		`Eq Ord`
`(a,b)`		`Eq Ord Bounded`

Definiowanie funkcji

W Haskellu funkcje zapisywane są w postaci równań. Na przykład funkcja zwracająca pole koła o promieniu r (wzór πr^2) może być zapisana następująco:

```
pole r = pi * r * r
```

Chociaż typ funkcji zostanie wywnioskowany automatycznie, zaleca się jawne określanie typu definiowanych funkcji:

```
pole :: (Floating a) => a -> a
pole r = pi * r * r
```

Litera `a` jest tzw. zmienną typową. Zapis typu funkcji składa się z co najwyżej trzech części: przed symbolem `::` umieszczamy nazwę funkcji, przed symbolem `=>` umieszczamy informację o klasie typów dotyczących zmiennych typowych, a na końcu umieszczamy zasadniczą informację o rodzaju odwzorowania korzystając z symbolu `->`. Widać więc, że funkcja `pole` dla argumentu będącego daną typu należącego do klasy `Floating` (jak np. `Float`) zwraca daną tego samego typu.

Typ funkcji wieloargumentowych zapisujemy za pomocą ciągu strzałek tak, jakby funkcja n-argumentowa mogła być traktowana jako jednoargumentowa zwracająca funkcję $(n-1)$-argumentową bądź traktowana jako dwuargumentowa zwracająca funkcję $(n-2)$-argumentową itd. Na przykład:

```
> let {dodaj :: (Num a) => a -> a -> a; dodaj x y = x+y}
> dodaj 1 2
3
> map (dodaj 10) [1..5]
[11,12,13,14,15]
```

Jeśli definicja funkcji zależy od wartości argumentu, to można ją zapisać na trzy sposoby. Po pierwsze za pomocą wyrażenia warunkowego:

```
ujemna :: (Num a) => a -> Bool
ujemna x = if signum x == -1 then True
           else False
```

Po drugie definicja funkcji może być wybrana za pomocą warunku boolowskiego:

```
g :: (Integral a) => a -> a
g n | n == 0    = 1
    | n == 1    = 2
    | otherwise = 2*g(n-1) + 3*g(n-2)
```

Po trzecie wreszcie, definicja funkcji może być wybrana na zasadzie dopasowania wzorca. Funkcja `g` z poprzedniego przykładu mogłaby być zdefiniowana również za pomocą wzorców, czyli kilku równań:

```
g :: (Integral a) => a -> a
g 0 = 1
g 1 = 2
g n = 2*g(n-1) + 3*g(n-2)
```

Wzorce można łączyć z warunkami wyboru, tak jak w poniższej funkcji zwracającej maksymalny element z niepustej listy:

```
maksymalny :: (Ord a) => [a] -> a
maksymalny [x] = x
maksymalny (x:y:zs) | x > y     = maksymalny (x:zs)
                    | otherwise = maksymalny (y:zs)
```

Gdy z danego wyrażenia korzystamy więcej niż jeden raz, to warto zdefiniować je w sekcji `where`, jak pokazano w poniższej definicji funkcji generującej wszystkie podzbiory zbioru reprezentowanego przez listę:

```
podzbiory :: [a] -> [[a]]
podzbiory []     = [[]]
podzbiory (x:xs) = [x:ys | ys <- yss] ++ yss
                   where yss = podzbiory xs
```

We wzorcach możemy używać znaku podkreślenia jako dopasowania do dowolnej wartości. Jest to szczególnie użyteczne, gdy interesuje nas samo występowanie danej, a nie jej wartość, na przykład:

```
drugi :: [t] -> t
drugi (_:x:_) = x
```

Oprócz funkcji, które mają nazwy, w Haskelu występują funkcje anonimowe. Ich składnia jest następująca:

```
\arg1 arg2 ... argK -> wyrażenie
```

Stosuje się je wszędzie tam, gdzie nazwa funkcji nie jest potrzebna, definicja jest prosta, a użycie jej skraca zapis wyrażenia. Szczególnie efektywne jest ich użycie w połączeniu z takimi funkcjami jak `foldl` i `foldr`. Są to funkcje typu:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

a ich zadaniem jest zredukowanie wszystkich elementów listy do jednej danej, za pomocą funkcji dwuargumentowej. Na przykład:

```
> foldl (\i1 i2 -> i1 ++ [head i2]) "" ["Ala", "Ewa", "Iza"]
"AEI"
```

Definiowanie własnych typów danych

W Haskellu dopuszcza się definiowanie nowych typów danych za pomocą konstruktorów umieszczonych w deklaracji `data`. Jako przykład rozważmy drzewa binarne przechowujące liczby całkowite

typu `Int`. Wierzchołek takiego drzewa zawiera liczbę oraz odsyłacze do drzewa lewego i prawego. Drzewo puste, nie zawierające żadnego wierzchołka, będziemy oznaczać słowem (konstruktorem bezparametrowym) `Nil`:

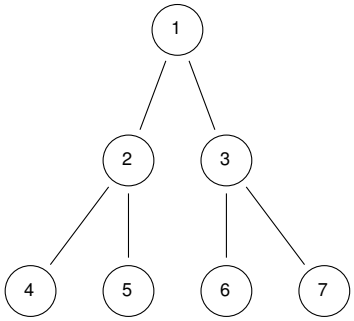
```
data Drzewo = Nil | Wierzcholek Int Drzewo Drzewo
```

Symbol `|` oznacza alternatywę — dane typu `Drzewo` mogą być tworzone albo za pomocą konstruktora `Nil`, albo za pomocą trójparametrowego konstruktora `Wierzcholek`.

Utworzenie wyrażenia reprezentującego dane drzewo binarne jest zadaniem trywialnym. Na przykład równanie:

```
d = Wierzcholek 1
      (Wierzcholek 2
        (Wierzcholek 4 Nil Nil)
        (Wierzcholek 5 Nil Nil)
      )
      (Wierzcholek 3
        (Wierzcholek 6 Nil Nil)
        (Wierzcholek 7 Nil Nil)
      )
```

definiuje drzewo przedstawione na rys. 32.



Rysunek 32: Drzewo binarne reprezentowane przez wyrażenie `d`

Równie łatwo jest definiować funkcje operujące na takich strukturach drzewiastych. Na przykład, wysokość drzewa jest wyznaczana przez następującą krótką funkcję:

```
wysokosc :: Drzewo -> Int
wysokosc Nil = -1
wysokosc (Wierzcholek _ dl dp) =
  1 + max (wysokosc dl) (wysokosc dp)
```

Programowanie obiektowe

Język Haskell, podobnie jak języki obiektowe, umożliwia tworzenie abstrakcyjnych klas, polimorfizm i hermetyzację.

Hermetyzacja danych jest realizowana przez umieszczenie każdego typu danych i funkcji operujących na danych tego typu w oddzielnym module. W każdym module można zaznaczyć, które z jego elementów będą widoczne na zewnątrz. Dzięki temu szczegóły implementacji mogą zostać ukryte wewnątrz modułu.

Polimorfizm jest realizowany przez zastosowanie klas typów. Klasy typów są w Haskellu zbiorem zasad, które musi spełnić typ danych, aby mógł być traktowany jako egzemplarz tej klasy. Można zdefiniować typ „Email” będący egzemplarzem klasy typów „Adres”. Wszystkie funkcje, które mogą zostać zastosowane w stosunku do zmiennej typu „Adres”, równie dobrze mogą zostać zastosowane do zmiennej typu „Email”.

Haskell umożliwia więc polimorfizm oraz hermetyzację. Nie pozwala jednak na grupowanie danych i funkcji w pojedynczy obiekt. Nie mamy więc do czynienia z takim dziedziczeniem danych jak

w języku C++ czy w Javie, niemniej dziedziczenie może zachodzić w obrębie klas typów.

Hermetyzacja

Każdy moduł zapisywany jest w osobnym pliku tekstowym, a jego struktura jest następująca

```
module Nazwa (...) where
...ciało-modułu...
```

Nazwa modułu musi być napisana z dużej litery i taką samą nazwę powinniśmy nadać plikowi z rozszerzeniem `.hs`, w którym moduł zapisujemy. Nawiasem `(...)` obejmuje się listę nazw funkcji i typów danych, z których użytkownik może korzystać. Można też tę część nagłówka modułu pominąć, wówczas wszystkie funkcje i typy danych będą dostępne. W skład ciała modułu wchodzą definicje klas typów, typów danych i funkcji.

Załóżmy, że w module o nazwie `M` zdefiniowano funkcje `f` i `g`, typy danych `A` z konstruktorami `Ka1`, `Ka2` i `Ka3` oraz typ danych `B` z konstruktorami `Kb1`, `Kb2` i `Kb3`. Jeśli na zewnątrz mają być widoczne: funkcja `f`, typ `A` ze wszystkimi konstruktorami, a typ `B` jedynie z konstruktorami `Kb1` i `Kb3`, to początek pliku z modułem powinien przedstawiać się następująco:

```
module M (A(..), B(Kb1,Kb3), f) where
...
```

Z modułu korzystamy w innych modułach po ich zaimportowaniu:

```
import Nazwa
```

Czasami dochodzi do kolizji nazw: w programie mamy funkcję lub typ o tej samej nazwie, jak w module. Wtedy należy użyć nazwy modułu, kropki i nazwy funkcji lub typu, wskazując, że chodzi nam o odpowiednią definicję zawartą w module.

Klasy typów i polimorfizm

Dzięki utworzeniu klasy typów (za pomocą słowa kluczowego `class`) mamy możliwość przeciążania operatorów i funkcji, co daje podstawy do realizacji polimorfizmu. Definicja klasy ma następującą strukturę:

```
class Nazwa-klasy zmienne-typowe where
  nazwa-funkcji/operatora :: typ-funkcji/operatora
  definicja-niektórych-funkcji/operatorów
```

Aby typ danych stał się egzemplarzem klasy, należy użyć konstrukcji:

```
instance Nazwa-klasy Nazwa-typu where
  przeciążenie-wymaganych-funkcji/operatorów
```

lub — w przypadku, gdy godzimy się na standardowy sposób przeciążenia wszystkich funkcji klasy — należy po definicji typu wymienić listę klas:

```
data definicja-typu deriving (lista-klas)
```

Jako przykład rozważmy definicję standardowej klasy `Eq` (umieszczoną w pliku `Prelude.hs`, który jest ładowany automatycznie):

```
class  Eq a  where
  (==), (/=) :: a -> a -> Bool

      -- Minimal complete definition:
      --      (==) or (/=)
  x /= y     =  not (x == y)
  x == y     =  not (x /= y)
```

Efekt wyprowadzenia klasy z innej klasy uzyskujemy określając jakiego rodzaju musi być zmienna typowa. Na przykład, standardowa klasa `Ord` została wyprowadzona z klasy `Eq`, co jest widoczne w jej nagłówku:

```
class  (Eq a) => Ord a  where
  compare              :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

      -- Minimal complete definition:
      --      (<=) or compare
  ...
```

Rozważmy uogólniony typ drzewa zdefiniowanego na str. 127, to znaczy taki, który umożliwi przechowywanie w strukturze drzewiastej danych innego typu (a nie tylko danych typu `Int`). Poniższy kod:

```
data (Eq a) => Drzewo a = Nil
                        | Wierzcholek a (Drzewo a) (Drzewo a)
  deriving Show

instance (Eq a) => Eq (Drzewo a) where
  Nil == Nil = True
  Nil == _   = False
  _ == Nil   = False
  Wierzcholek x lx px == Wierzcholek y ly py =
    and [x == y, lx == ly, px == py]
```

czyni typ `Drzewo` egzemplarzem klasy `Eq`, przeciążając operator porównania w ten sposób, że dwa drzewa są równe wtedy, gdy obydwa są identyczne (mają taką samą postać i wartości w odpowiadających sobie wierzchołkach). Dodatkowo typ `Drzewo` został „wyprowadzony” z klasy `Show`. Tym samym możliwe jest uzyskiwanie tekstowej reprezentacji drzew.

Warto dodać, że obok zapisanych w języku wymogów na egzemplarze (czyli podanie implementacji pewnych operacji) często klasy są powiązane z nieformalnie określonymi oczekiwaniami względem egzemplarzy. Na przykład `(==)` powinien być relacją równoważności albo implementacja operatora `(<=)` powinna odpowiadać naturalnemu porządkowi wśród danych określonego typu.

Przykłady

Rozważmy grę logiczną dla dwóch osób, dalej zwanych graczami A i B, w której obydwaj gracze wykonują ruchy na przemian. W tej grze nie ma remisów, czyli wygrywa A lub B. Załóżmy dalej, że liczba możliwych ruchów, które można wykonać w danej chwili, zależy tylko od bieżącej pozycji, a nie zależy od pozycji, które wystąpiły wcześniej w grze. Każda rozgrywka jest zatem ciągiem par konfiguracja-gracz (sytuacji), gdzie „konfiguracja" oznaczać może: bieżący układ pionów, ostatnio odkrytą kartę itd., natomiast „gracz" oznacza gracza, na którego przypada ruch. Gdy rozpoczyna grę A, to rozgrywką jest ciąg: $K_0\ A$, $K_1\ B$, aż do pewnej sytuacji $K_n\ A$ lub $K_n\ B$, która jest sytuacją końcową, a rozgrywkę wygrał jeden z graczy.

Dla tak zdefiniowanej gry można napisać ogólną (polimorficzną) funkcję sprawdzającą, czy gracz na którego przypada ruch może doprowadzić do zwycięstwa. Sytuację $K_i\ G$ może wygrać gracz G ($G \in \{A, B\}$) jeśli jest to już sytuacja końcowa wygrana przez G lub jeśli istnieje przynajmniej jeden ruch po którym powstanie sytuacja, z której przeciwnik nie ma kontynuacji wygrywającej.

Definicja klasy opisującej sytuację w naszej abstrakcyjnej grze oraz implementacja sprawdzania możliwości zwycięstwa zawarte są poniżej:

```
module Sytuacja where

data Gracz = A | B deriving (Eq, Show)

class Sytuacja a where
  kolejne    :: a -> [a]
  koncowa    :: a -> Bool
  kto_wygral :: a -> Gracz
  czyj_ruch  :: a -> Gracz
  koncowa x = null $ kolejne x

mozna_wygrac :: Sytuacja a => a -> Bool
mozna_wygrac x =
  if koncowa x then
    kto_wygral x == czyj_ruch x
  else
    or [not (mozna_wygrac y) | y <- kolejne x]

przeciwnik :: Gracz -> Gracz
przeciwnik A = B
```

```
przeciwnik B = A
```

Proszę zwrócić uwagę, że sytuację, w której nie można już wykonać żadnego ruchu (`null $ kolejne x`, czyli inaczej `kolejne x == []`), uznajemy za końcową. Jeśli zgodnie z zasadami gry, jej zakończenie może nastąpić wcześniej, to funkcję `koncowa` należy przedefiniować w typie opisującym tę konkretną grę.

Nim jest przykładem gry spełniającej podane wyżej założenia. Zasady są następujące. Rozpoczynamy od ustalonej liczby całkowitej dodatniej, powiedzmy 29. Gracz, na którego przypada ruch może od bieżącej liczby odjąć 1, 2 lub 3, oby tylko nowa liczba była większa lub równa 0. Wygrywa ten z graczy, który nie może wykonać ruchu. Aby można było dla gry Nim skorzystać z funkcji `mozna_wygrac`, wystarczy zaimplementować trzy funkcje z klasy `Sytuacja`:

```
module Nim where

import Sytuacja

data Nim = N Int Gracz deriving Show

instance Sytuacja Nim where
  kolejne (N i g) =
    [N i' (przeciwnik g) | i' <- [i-3..i-1],
                           i' >= 0]
  kto_wygral (N _ g) = g
  czyj_ruch (N _ g) = g
```

Odpowiedzią na pytanie, czy w grze Nim[28] zwycięża ten, kto zaczyna, będzie wartość wyrażenia:

```
*Nim> mozna_wygrac $ N 29 A
False
```

[28] Gra Nim jest bardzo dobrze rozpracowana w teorii gier kombinatorycznych, dlatego akurat dla niej można opracować inny, efektywniejszy algorytm.

Moduł `Sytuacja` może być wykorzystany do bardziej skomplikowanej gry niż Nim, na przykład do „wilka i owiec". Jest to tradycyjna gra o typie warcabowym. Jeden z graczy dysponuje jednym pionem czarnym (wilkiem), drugi — czterema pionami białymi (owcami). Przed rozpoczęciem gry owce obsadzają pierwszy rząd swojej połowy szachownicy, wilk staje na jednym dowolnie wybranym polu swego krańcowego rzędu szachownicy.

Cel gry jest inny dla gracza grającego owcami, inny dla gracza grającego wilkiem. Gracz grający wilkiem wygrywa, gdy swoim pionem przedostanie się na pierwszy rząd pól szachownicy po stronie owiec. Gracz grający owcami wygrywa w chwili, gdy uda mu się zablokować wilka tak, żeby nie mógł on wykonać przypadającego nań posunięcia.

Grę rozpoczyna gracz grający wilkiem. W jednym ruchu pion tego gracza może przejść o jedno pole po przekątnej w dowolną stronę. Piony gracza grającego owcami w kolejnych ruchach przechodzą także o jedno pole po przekątnej, lecz tylko do przodu (z dołu

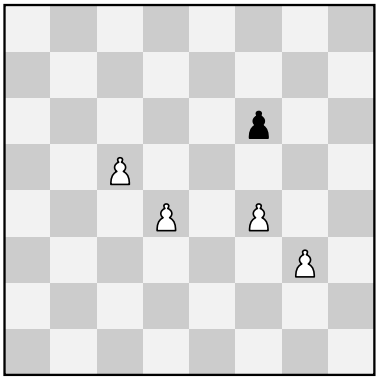
planszy do góry). Wilk może się cofać, owce nie mogą się cofać. Implementacja tej gry została przedstawiona poniżej.

```
module WiO where

import Sytuacja

data WiO = W [(Int,Int)] Gracz deriving Show

instance Sytuacja WiO where
  kolejne (W p@[(wx,wy),o1,o2,o3,o4] A) =
    [W ((wx',wy'):drop 1 p) B | wx' <- [wx+1, wx-1],
                                wy' <- [wy+1, wy-1],
                                wx' >= 1, wx' <= 8,
                                wy' >= 1, wy' <= 8,
                                notElem (wx',wy') p]
  kolejne (W p@[w,o1,o2,o3,o4] B) =
    [W [w,(x,y),o2,o3,o4] A | y <- [snd o1 + 1, snd o1 - 1],
                              x <- [fst o1 - 1],
                              x >= 1, x <= 8,
                              y >= 1, y <= 8,
                              notElem (x,y) p] ++
    [W [w,o1,(x,y),o3,o4] A | y <- [snd o2 + 1, snd o2 - 1],
                              x <- [fst o2 - 1],
                              x >= 1, x <= 8,
                              y >= 1, y <= 8,
                              notElem (x,y) p] ++
    [W [w,o1,o2,(x,y),o4] A | y <- [snd o3 + 1, snd o3 - 1],
                              x <- [fst o3 - 1],
                              x >= 1, x <= 8,
                              y >= 1, y <= 8,
                              notElem (x,y) p] ++
    [W [w,o1,o2,o3,(x,y)] A | y <- [snd o4 + 1, snd o4 - 1],
                              x <- [fst o4 - 1],
                              x >= 1, x <= 8,
                              y >= 1, y <= 8,
                              notElem (x,y) p]
  koncowa (W p@[(wx,wy),o1,o2,o3,o4] g) =
    (null $ kolejne (W p g)) || (wx == 8)
  kto_wygral (W p g) =
    if null $ kolejne (W p g) then przeciwnik g
    else A
  czyj_ruch (W _ g) = g
```

Sytuacja w grze wilk i owce, `W t g`, reprezentowana jest przez listę `t` współrzędnych wilka i czterech owiec oraz gracza `g` będącego albo `A` (wilkiem) albo `B` (owcą). Zgodnie z regułami sytuacją kończącą grę jest niemożność wykonania ruchu lub ustawienie wilka na dolnym polu: odpowiedzialna za sprawdzenie tego jest funkcja o nazwie `koncowa`. Rzędy pól planszy numerowane są od góry do

dołu od 1 do 8, natomiast kolumny od lewej do prawej, także od 1 do 8. Aby zatem sprawdzić, czy w sytuacji z rys. 33, wilk może wyjść z opresji, należy wyznaczyć wartość wyrażenia:

```
*WiO> mozna_wygrac $ W [(3,6),(6,7),(4,3),(5,4),(5,6)] A
False
```



Rysunek 33: Sytuacja z gry wilk i owce

Podsumowanie

W rozdziale tym przedstawiono ogólną informację o języku Haskell, po czym omówiono koncepcję programowania obiektowego w tym języku. Już na podstawie tego — z konieczności krótkiego — przeglądu konstrukcji języka, mogliśmy się zorientować, że Haskell zawiera trzy zasadnicze komponenty programowania funkcyjnego: (a) zbiór typów danych, w tym struktur wyższego poziomu takich jak listy; (b) zbiór funkcji standardowych (ang. built-in functions) służących do manipulowania danymi podstawowych typów; (c) zbiór funkcji wyższego rzędu (ang. high-order functions), które służą do dostarczenia ogólnych realizacji algorytmów, zaś programista parametryzuje je własnymi funkcjami. Przykładem funkcji wyższego rzędu są omówione w treści rozdziału funkcje: `map`, `filter` lub `foldl`.

Nie ulega jednak żadnej wątpliwości, że do rosnącej popularności języka Haskell przyczyniła się możliwość realizacji paradygmatu programowania obiektowego w tym języku. W drugiej części rozdziału widzieliśmy, jak dane i funkcje mogą być hermetyzowane w modułach (ang. encapsulation, czyli kapsułkowanie, ukrywanie informacji), w jaki sposób można tworzyć hierarchię klas (ang. inheritance, czyli dziedziczenie) i wreszcie, jak kolekcje obiektów mogą dotyczyć danych różnego typu (zob. także program z zad. 7.), a wywołanie funkcji powoduje zachowanie odpowiednie dla danego typu obiektu wywoływanego (ang. polymorphism, czyli wielopostaciowość).

Warto pamiętać, że implementację typowych operacji na takich strukturach danych, jak tablice, łańuchy, zbiory, mapy i drzewa zawarto w dodatkowych bibliotekach, dostępnych na oficjalnej internetowej stronie języka Haskell[29].

[29] `http://www.haskell.org/ghc/docs/-latest/html/libraries/index.html`

Zadania

Zad. 1. Utwórz niżej przedstawione listy za pomocą możliwie najkrótszej definicji:

```
[1,2,3,4,5,6,7,8,10,11,12,13,14,15]
[2,-3,4,-5,6,-7,8,-9,10,-11,12,-13,14,-15]
```

Zad. 2. Zdefiniuj funkcję typu:

```
r :: Int -> [Int]
```

która dla całkowitego n zwraca listę składającą się z jednej 1, dwóch 2 itd., aż do n:

```
> r 4
[1,2,2,3,3,3,4,4,4,4]
```

Zad. 3. Napisz funkcję

```
f :: [Int] -> Int -> [Int]
```

która dla listy liczb całkowitych oraz liczby całkowitej zwraca listę indeksów określających miejsce występowania liczby w liście.

Zad. 4. Napisz funkcję, która dla podanych dwóch list zwraca `True`, gdy wszystkie elementy pierwszej listy występują także w drugiej liście lub zwraca `False` w przeciwnym razie. Funkcja ta powinna być typu:

```
zawiera :: Eq a => [a] -> [a] -> Bool
```

Zad. 5. Przyjmując definicję drzewa podaną na str. 127, napisz funkcję

```
wywazone :: [Int] -> Drzewo
```

tworzącą dokładnie wyważone drzewo na podstawie listy liczb całkowitych. Drzewo jest dokładnie wyważone, jeżeli dla każdego wierzchołka, liczby wierzchołków w jego lewym i prawym poddrzewie różnią się co najwyżej o 1.

Zad. 6. Jak w matematyce nazywamy wynik działania funkcji:

```
f :: Int -> Int -> [[Int]]
f n k | k > n     = []
      | k == 1    = [[n]]
      | otherwise = [i:xs | i  <- [1..n-k+1],
                            xs <- f (n-i) (k-1)]
```

dla pary liczb n i k?

Zad. 7. Zdefiniujmy klasę typów kontenerowych, których zadaniem będzie przechowywanie danych wybranego typu. Do podstawowych operacji wykonywanych na kontenerze zaliczymy:

- utworzenie pustego kontenera (funkcja `pusty`),
- dodanie nowego elementu (funkcja `dodaj`),
- usunięcie jednego elementu (funkcja `usun`),
- zwrócenie pierwszego elementu, gdzie kolejność determinowana jest rodzajem kontenera (funkcja `pierwszy`),
- sprawdzenie, czy kontener jest pusty (funkcja `czyPusty`).

Zakładając, że kontener może przechowywać elementy dowolnego typu (ale tego samego typu w obrębie jednego kontenera), definicja stosownej klasy może być taka, jak w programie poniżej.

Praktyczną realizacją kontenera może być stos lub kolejka. W przypadku stosu kolejność usuwania elementów jest odwrotna do kolejności ich wstawienia (ang. LIFO — last in first out). Elementy mogą być przechowywane w zwykłej liście. W przypadku kolejki kolejność usuwania elementów jest dokładnie taka sama jak kolejność ich wstawienia (ang. FIFO — first in first out). Aby zminimalizować czasy realizacji operacji kontenerowych zdecydowano się reprezentować kolejkę za pomocą dwóch list. Jedna z list przechowuje początkowe elementy kolejki, a druga całą resztę, ale w odwrotnej kolejności.

```
module Kontener ( Stos(),
                  Kolejka(),
                  pusty,
                  dodaj,
                  usun,
                  pierwszy,
                  czyPusty
                ) where

class Kontener k a where
  pusty    :: k a
  dodaj    :: a -> k a -> k a
  usun     :: k a -> k a
  pierwszy :: k a -> a
  czyPusty :: k a -> Bool

data Stos a = S [a] deriving Show

instance Kontener Stos a where
  pusty              = S []
  dodaj x (S xs)     = S (x:xs)
  usun (S (_:xs))    = S xs
  pierwszy (S (x:_)) = x
  czyPusty (S [])    = True
  czyPusty (S _ )    = False

data Kolejka a = Q ([a],[a]) deriving Show

instance Kontener Kolejka a where
  pusty                  = Q ([],[])
  dodaj x (Q ([],[]))    = Q ([x],[])
  dodaj y (Q (xs,ys))    = Q (xs,y:ys)
  usun (Q ([],ys))       = Q (tail (reverse ys), [])
  usun (Q (x:xs,ys))     = Q (xs,ys)
  pierwszy (Q ([],ys))   = last ys
```

```
  pierwszy (Q (x:xs,ys)) = x
  czyPusty (Q ([],[]))   = True
  czyPusty _             = False
```

Poniższy program:

```
import Kontener

main = do
  putStrLn $ show $
    dodaj 1 (dodaj 2 (dodaj 3 (pusty::(Stos Int))))
  putStrLn $ show $ dodaj "Ala"
    (dodaj "ma" (dodaj "kota" (pusty::(Kolejka String))))
```

zapisano w pliku `Zad.hs`, a następnie skompilowano za pomocą polecenia:

```
$ ghc -fglasgow-exts --make Zad
```

Jaki będzie efekt wykonania tego programu?

Zad. 8. Przyjmij definicje klasy kontenera, typu stosowego i kolejkowego przedstawione w poprzednim zadaniu. Zaimplementuj funkcję polimorficzną usuwającą wszystkie elementy kontenera oprócz ostatniego, jak pokazano na przykładzie:

```
> doOstatniego $ dodaj 3 (dodaj 2 (dodaj 1 (pusty::(Stos Int))))
S [1]
> doOstatniego $ dodaj 2 (dodaj 1 (pusty::(Kolejka Int)))
Q ([],[2])
```

Zad. 9. Zdefiniuj typ zbiorowy implementujący funkcje klasy `Kontener` (p. zad. 7):

```
instance (Ord a) => Kontener Zbior a where
  ...
```

`Zbior` powinien charakteryzować się następującymi cechami:

- elementy w zbiorze nie mogą się powtarzać,
- `pierwszy` element to taki, który jest mniejszy od każdego innego elementu znajdującego się w zbiorze.

Zad. 10. Niech, dla $n = 0, 1, 2, \ldots$, ciąg $f(n)$ będzie rosnącym ciągiem liczb całkowitych. Dla takiego ciągu można zbudować tablicę różnic, w której wartość na każdej pozycji jest różnicą dwóch liczb znajdujących się bezpośrednio nad nią („prawa" odjąć „lewa"):

```
n =  0   1   2   3   4   5   6
     a   ?   ?   ?   ?   ?   ?
       b   ?   ?   ?   ?   ?
         c   ?   ?   ?   ?
           d   d   d   d
```

Pierwszy wiersz: $f(0) = a,\ f(1),\ f(2), \ldots$, składa się z wartości ciągu; niżej mamy pierwsze różnice: $f(1) - a = b,\ f(2) - f(1), \ldots$; jeszcze niżej drugie różnice itd. Jeśli wszystkie k-te różnice są równe, to $f(n)$ jest wielomianem stopnia k. Jego wzór ma postać:

$$a \cdot \binom{n}{0} + b \cdot \binom{n}{1} + c \cdot \binom{n}{2} + d \cdot \binom{n}{3}.$$

Zdefiniuj funkcję określającą stopień wielomianu na podstawie nieskończonej listy jego wartości:

```
> st [n^4 + 5 | n <- [0..]]
4
> st [1,7..]
1
> let f n = if n<=1 then 4*n else 2 * f (n-1) - f (n-2) + n
> st [f i | i <- [0..]]
3
```

Python

Ogólne informacje o języku

Python jest językiem interpretowanym, z którego możemy korzystać na dwa sposoby: interaktywnie — wpisując instrukcje i uzyskując natychmiastowy efekt ich działania oraz wsadowo — wykonując cały program zapisany w pliku tekstowym. Interpreter tego języka[30], opis sposobu instalacji oraz dokumentację wraz z formalnymi definicjami można pobrać ze strony `www.python.org`.

Przyjmuje się zwyczajowo, że pliki źródłowe z programami w Pythonie zapisywane są pod nazwą z rozszerzeniem `.py`. Rozpoczęcie pracy interaktywnej rozpoczyna się po wydaniu polecenia `python`, natomiast wykonanie programu zapisanego w pliku `nazwa.py` rozpocznie się po wydaniu polecenia

```
python nazwa.py
```

Podobnie jak w języku Java, tworzony jest kod pośredni, jednak w Pythonie uruchomienie tego kodu pośredniego (`nazwa.pyc`) dzieje się poza kontrolą użytkownika.

Python jest językiem, który pozwala równie łatwo realizować paradygmat programowania strukturalnego (proceduralnego) jak i obiektowego. Układ typowego programu jest taki sam jak w większości języków programowania: załadowanie potrzebnych bibliotek (za pomocą instrukcji `import` lub `from`), definicje podprogramów, definicje klas, główna część obliczeniowa. Nie ma sensu przedstawiania takiego układu w sposób formalny, bo w Pythonie mamy do czynienia z daleko idącą dowolnością dotyczącą struktury programu; np. instrukcje mogą być umieszczone pomiędzy definicjami funkcji, a załadowanie biblioteki — w ciele funkcji. Interpreter wykonuje kod programu, jaki napotyka idąc z góry na dół, od pierwszego wiersza. Warto także pamiętać o następujących cechach wyróżniających Pythona na tle innych języków:

- brak deklaracji typu zmiennych,
- brak średników oddzielających instrukcje — instrukcje umieszczane są w kolejnych wierszach programu (pliku tekstowego),
- obowiązkowe wcięcia pełniące rolę pascalowych słów kluczowych „begin” i „end” (lub „`{`” i „`}`” znanych z C++ i Javy)

[30] Opis Pythona przedstawiony w niniejszym rozdziale dotyczy wersji 2.6 tego języka. Obecnie, równolegle do kolejnych wersji 2.x, opracowywane są także wersje 3.x, które niestety nie są do końca zgodne z 2.x. Różnice są opisane na internetowej stronie Pythona. Zdecydowaliśmy się scharakteryzować syntaktykę i semantykę języka Python wg. dokumentacji do wersji 2.6, ponieważ przytłaczająca liczba opracowanych bibliotek nie jest kompatybilna z wersjami 3.x.

- przenośność kodu: programy mogą działać pod kontrolą takich systemów operacyjnych jak Linux, Mac, MS-DOS, Windows, Windows NT, OS/2 i in.,
- możliwość potraktowania łańcucha znakowego jako programu i wykonanie go (funkcje `eval` i `exec`),
- możliwość wyprowadzania klas z wbudowanych typów,
- trzy rodzaje składowych klasy: atrybuty (pola), właściwości i metody,
- możliwość dynamicznego dodawania nowych atrybutów do obiektów,
- funkcje i obiekty mogą pełnić rolę iteratorów (instrukcja `yield`, metoda `__iter__(self)`).

Wbudowane typy i struktury danych

Oprócz wbudowanych typów danych wymienionych i scharakteryzowanych w tabeli 5 warto pamiętać o typie `Decimal`, dostępnym po załadowaniu biblioteki `decimal`, który umożliwia wykonywanie operacji zmiennoprzecinkowych o dowolnie dużej dokładności:

```
>>> from decimal import *
>>> getcontext().prec = 30
>>> Decimal(1) / Decimal(3)
Decimal('0.333333333333333333333333333333')
```

W Pythonie nawet zmienne tak podstawowych typów jak typy liczbowe (np. `int`, `float`) są obiektami odpowiednich klas. Jednakże sposób ich użycia jest tak samo intuicyjny jak w innych językach programowania. Należy jednak w tym miejscu podkreślić, że w Pythonie wszelkie odwołania odbywają się przez referencję. Na przykład zapis:

```
>>> a = 12
>>> b = "Ala"
```

oznacza, że `a` jest referencją do obiektu klasy `int` przechowującego wartość 12, natomiast `b` jest referencją do obiektu klasy `str` przechowującego łańcuch `'Ala'`. Powyższe dwie instrukcje są zatem równoważne następującemu zapisowi w języku Java:

```
Integer a = new Integer(12);
String b = new String("Ala");
```

i następującemu zapisowi w języku C++:

```
int liczba = 12; int &a = liczba;
const string b("Ala");
```

Ważne jest rozróżnienie między typami niezmiennymi i takimi, które można modyfikować. Spośród typów wymienionych w tabeli 5 modyfikować można zmienne typów: `list`, `dict`, `file` i `set`. Można zatem zmienić wybrany element listy, ale nie krotki:

Tabela 5: Wbudowane typy i struktury danych

Nazwa	Opis	Przykład użycia
`object`	rodzic wszystkich typów i klas	`class Moja(object):`
`NoneType`	tego typu jest obiekt null	`x == None`
`bool`	typ boolowski	`True != False`
`int`	liczba całkowita	`x = 2`
`long`	liczba całkowita o nieograniczonym zakresie	`x = 1009872116134L`
`float`	liczba zmiennoprzecinkowa	`x = 2.123`
`complex`	liczba zespolona	`z = (1+2j)*(0.3-4.1j)`
`str`	łańcuch znaków	`x = "abc ew"`
`unicode`	łańcuch znaków Unicode	`x = u"śląski"`
`list`	lista (tablica) elementów, niekoniecznie tego samego typu	`x = [1, 2, 3]`
`tuple`	krotka, czyli niezmienna lista	`x = (1, 2, 3)`
`xrange`	iterator; jego rozmiar nie zależy od długości sekwencji	`for i in xrange(9):`
`dict`	słownik, ze stałym czasem dostępu do elementów	`s = {'A':19, 'Bc':21}`
`file`	plik	`pk = open("d.txt","r")`
`set`	zbiór elementów (niekoniecznie tego samego typu)	`z = set([1.3, 'a'])`
`frozenset`	niezmienny zbiór elementów	`z = frozenset(['a', 2])`

```
>>> tab = [3, 4, 5]
>>> tab[0] += 10
>>> tab
[13, 4, 5]
>>> k = (3, 4, 5)
>>> k[0] += 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Chociaż efekt działania poniższych instrukcji przypisania jest taki sam jak w innych językach programowania:

```
>>> x = 2
>>> x += 1
>>> x
3
```

warto zdać sobie sprawę, że najpierw zmienna x była referencją do obiektu 2, a na końcu była referencją do innego obiektu, 3, który powstał w wyniku wykonania instrukcji `x += 1`. Dlatego też mamy dwa sposoby porównywania: porównywanie „wartości" za pomocą operatora == i porównywanie „identyczności" za pomocą słowa kluczowego `is`. Dwie zmienne są identyczne jeśli odwołują się do tego samego obiektu:

```
>>> a = set([1])
>>> b = set(range(1,20))
>>> c = set(range(2,20))
>>> a == b-c
True
>>> a is b-c
False
```

Aby sprawdzić, czy dwie zmienne są referencją do tego samego obiektu, można także wykorzystać funkcję `id(x)`, która dla podanego obiektu `x` zwraca jego niepowtarzalny numer — liczbę całkowitą.

Każda dana przechowywana i wykorzystywana w Pythonie jest obiektem, więc instrukcja przypisania `x = y` nie dokonuje skopiowania obiektu `y`. Aby `x` stał się kopią `y` należy skorzystać z odpowiedniej metody (jeśli `y` takową posiada), np.:

```
x = y.copy()
```

czy też z funkcji `copy()` lub `deepcopy()` dostępnych w module `copy`.

Na prostych przykładach zilustrujemy użycie większości z wymienionych typów danych. Będzie to także okazja do zaprezentowania wybranych funkcji języka Python.

O tym, czy liczba całkowita zostanie zapisana jako obiekt `int`, czy jako `long` decyduje przyrostek `L`, bądź dostatecznie duży wynik operacji:

```
>>> a = 24
```

```
>>> type(a)
<type 'int'>
>>> b = 13**a
>>> print b
5428007703743705127715953 61
>>> type(b)
<type 'long'>
>>> a = 24L
>>> type(a)
<type 'long'>
```

W Pythonie większość operatorów zapisywana jest za pomocą tych samych symboli co w języku C++ lub Java. Użyty w przykładzie operator `**` oznacza potęgowanie.

Podczas zapisywania liczb zespolonych obowiązuje składnia *część_rzeczywista* + *część_urojona*`j`. Liczba taka przechowywana jest jako para liczb zmiennoprzecinkowych:

```
>>> a = 2.0 + 3.0j
>>> dir(a)
['__abs__', '__add__', ... , 'imag', 'real']
>>> a.imag
3.0
```

Za pomocą funkcji `dir` uzyskaliśmy listę składowych obiektu `a`. Jeśli funkcja `dir` zostałaby wywołana bez argumentów, to jako wynik otrzymalibyśmy listę zawierającą wszystkie nazwy zdefiniowane w interakcyjnej przestrzeni nazw.

Dostęp do poszczególnych znaków łańcucha — podobnie jak dostęp do elementów listy — odbywa się za pomocą indeksowania nawiasami kwadratowymi. Należy jednak pamiętać, że obiekty klas `str` i `unicode` są niezmienne. Nie można zatem ani usuwać, ani dodawać, ani zmieniać liter. Liczba metod służących do przetwarzania łańcuchów jest duża i z łatwością można ją znaleźć w dokumentacji.

Funkcja `map` jest przedstawicielką klasy funkcji wbudowanych języka Python, używanych w programowaniu funkcyjnym. Pierwszym jej argumentem jest funkcja, która będzie wywoływana z kolejnymi elementami iterowalnego drugiego argumentu. Wynikiem jest lista obiektów zwróconych przez kolejne wywołania, np.

```
>>> napis = "Ala ma kota"
>>> map(len, napis.split())
[3, 2, 4]
```

Pokrewnymi funkcjami zaczerpniętymi z programowania funkcyjnego są `filter` (do odrzucania elementów nie spełniających jakiegoś warunku) oraz `reduce` (do realizacji operacji $(\cdots(((a_0 + a_1) + a_2) + a_3) + \cdots + a_n)$ na tablicy a, gdzie zamiast operatora dodawania może być inna funkcja dwuargumentowa). Metoda `split(separator)` klasy `str` zwraca listę słów, które w danym łańcuchu oddzielone są łańcuchem `separator`.

Elementami list (tablic) mogą być obiekty różnych typów. Oprócz standardowego sposobu indeksowania mamy także dostępne wycinanie oraz indeksowanie liczbami ujemnymi (dotyczy to także łańcuchów):

```
>>> tab = range(3, 11)
>>> print tab
[3, 4, 5, 6, 7, 8, 9, 10]
>>> tab[0], tab[-1], tab[1:4], tab[5:], tab[2:-2]
(3, 10, [4, 5, 6], [8, 9, 10], [5, 6, 7, 8])
```

Poza wykorzystaniem funkcji `range`, inny wygodny sposób tworzenia listy znany jest w programowaniu funkcyjnym jako *list comprehension*:

```
>>> [i for i in xrange(400) if unichr(i) in u"ąęćńóśźż"]
[243, 261, 263, 281, 324, 347, 378, 380]
```

W ten sposób uzyskaliśmy tablicę kodów wybranych znaków diakrytycznych. Część warunkowa nie jest wymagana. Na przykład:

```
>>> [x*x for x in xrange(200, 207)]
```

wygeneruje tablicę kwadratów liczb od 200 do 206. Różnica między funkcją `range` i `xrange` jest taka, że ta pierwsza zwraca obiekt klasy `list`, a druga zwraca obiekt klasy `xrange`. Wszędzie tam, gdzie to możliwe zaleca się stosowanie funkcji `xrange`, gdyż lista nie jest wtedy tworzona w pamięci komputera.

Listy można w różny sposób modyfikować. Za pomocą `tab.append(x)` na końcu listy `tab` dodajemy element `x`; wstawiamy elementy za pomocą metody `insert`; usuwamy jeden lub więcej elementów za pomocą operatora `del`; porządkujemy elementy (sortowanie w miejscu) za pomocą metody `sort`.

Słowniki (obiekty klasy `dict`) służą do przechowywania par klucz-wartość. Ich zaletą jest bardzo szybki czas dostępu do elementu o zadanym kluczu niezależnie od wielkości słownika. Uzyskuje się to dzięki wykorzystaniu funkcji mieszających. Wobec tego kluczem słownika może być dowolny obiekt `ob`, dla którego można uzyskać liczbę całkowitą za pomocą funkcji `hash(ob)`. Obiektami takimi są wszystkie wbudowane typy niezmienne, obiekty klas zdefiniowanych przez programistę (wartością funkcji mieszającej jest wówczas ich `id()`) oraz wszystkie obiekty, które mają zdefiniowaną metodę `__hash__()` i do porównywania `__eq__()` lub `__cmp__()`. Najprostszą metodą utworzenie słownika jest wpisanie par klucz:wartość pomiędzy znaki `{` oraz `}`:

```
>>> s = {('A', 4): 'skoczek', ('B', 6): 'goniec'}
```

taki sam słownik uzyskamy również za pomocą następujących instrukcji:

```
>>> s = {}
>>> s[('A', 4)] = 'skoczek'
>>> s[('B', 6)] = 'goniec'
```

albo

```
>>> s = dict(zip([('A', 4), ('B', 6)], ['skoczek', 'goniec']))
```

Jeśli klucze są łańcuchami, to słownik może być utworzony w jeszcze prostszy sposób:

```
>>> dni = dict(pon=1, wt=2, sr=3, czw=4, pt=5)
>>> print dni
{'pon': 1, 'sr': 3, 'wt': 2, 'czw': 4, 'pt': 5}
```

Często powtarzanymi operacjami na słownikach są: sprawdzenie istnienia klucza (`klucz in s` lub `klucz not in s`), zmiana wartości skojarzonej z kluczem (`s[klucz] = nowa`). Do wykonania pewnej operacji z wszystkimi elementami słownika przydatna jest metoda `iteritems`:

```
>>> for (klucz, wart) in s.iteritems():
...   print wart, "na", klucz
...
goniec na ('B', 6)
skoczek na ('A', 4)
```

Podobnie jak z list do usuwania elementów ze słowników służy operator `del`.

Zbiory reprezentują kolekcję niepowtarzalnych obiektów. Istnieją dwa podstawowe typy zbiorów: zmienny `set` oraz niezmienny `frozenset` (zbiory niezmienne nie mogą być modyfikowane po utworzeniu). Elementami zbiorów mogą być wszystkie „haszowalne" obiekty; obowiązują więc te same wymagania, które dotyczą kluczy w słowniku. Do najczęściej wykonywanych operacji na zbiorze `z` typu `set` i elemencie `x` należą: sprawdzenie przynależności (`x in z`), dodanie elementu do zbioru (`z.add(x)`), usunięcie elementu ze zbioru (`z.remove(x)` lub `z.discard(x)`). Dla dwóch dowolnych zbiorów `u` i `v` możemy uzyskać zbiór, który jest ich sumą mnogościową (`u | v`), przecięciem (`u & v`), różnicą (`u - v`) i różnicą symetryczną (`u ^ v`). Prawdziwość zawierania się zbiorów sprawdzamy za pomocą operatorów `<=`, `>=`, `<` i `>`.

Do wbudowanych funkcji wspólnych dla wszystkich typów sekwencyjnych należą: `len(s)` — zwracająca liczbę elementów w sekwencji `s`, `all(s)` — zwracająca `True`, jeśli wszystkie elementy w `s` mają wartość logiczną `True`, `any(s)` — zwracająca `True`, jeśli dowolny element w `s` ma wartość logiczną `True`. Warto przy tym pamiętać, że wszystkie niżej wymienione wartości są równoważne logicznemu `False`:

- `None`,
- `False`,
- zero dowolnego typu numerycznego, np.: `0`, `0L`, `0.0`, `0j`,
- pusta sekwencja, np.: `''`, `()`, `[]`,

- pusty słownik,
- obiekty klas zdefiniowanych przez użytkownika, jeśli zdefiniowano metodę `__nonzero__()` lub metodę `__len()__`, kiedy ta metoda zwraca całkowite zero lub boolowskie `False`.

Wszystkie inne wartości są równoważne logicznemu `True`.

Podstawowe instrukcje, funkcje i wyjątki

Podrozdział ten rozpoczniemy od przykładu, który zilustruje tezę, że połączenie wbudowanych typów, elementów programowania funkcyjnego oraz prostej składni instrukcji Pythona, pozwala na bardzo zwięzłe zapisywanie algorytmów. Idea sortowania szybkiego — zakładając, że mamy do czynienia z tablicą `x` różnych liczb — może być zrealizowana w niżej przedstawiony sposób. Znak kontynuacji wiersza (\) oznacza, że wiersz ma swój ciąg dalszy.

```
def qsort(x):
  if len(x) <= 1: return x
  else: return qsort([a for a in x if a < x[0]]) + \
               [x[0]] + \
               qsort([a for a in x if a > x[0]])
```

Co do złożoności pamięciowej, taki sposób zakodowania algorytmu „quicksort” pozostawia wiele do życzenia, unaocznia on jednakże jak zwarty, krótki i czytelny może być program w języku Python.

Instrukcja warunkowa, przykład użycia której widzieliśmy w funkcji `qsort`, ma następującą składnię:

```
if wyrażenie1:
  sekcja1
elif wyrażenie2:
  sekcja2
else:
  sekcja3
```

Na wstępie niniejszego rozdziału podkreślono, że w Pythonie nie ma znaczników ograniczających, a ich rolę pełnią wcięcia. Umieszczenie dwukropka po każdym z wyrażeń spełniających funkcję warunków oraz po słowie kluczowym `else` jest obowiązkowe. Sekcji `elif` może być dowolnie dużo. Najczęściej używanymi operatorami w warunkach są:

- porównania: `<`, `<=`, `==`, `!=`, `>=`, `>`,
- identyczności: `is`, `is not`,
- przynależności: `in`, `not in`,
- logiczne: `not`, `and`, `or`.

Dopuszczalne jest łączenie porównań w sposób przypominający zapis matematyczny:

```
if 1 <= x <= 10:
  print "tak"
```

Istnieje też wyrażenie warunkowe którego pierwowzorem było wyrażenie `a?b:c` znane z języka C++. Jego składnia w Pythonie jest następująca:

```
rezultatGdyTrue if wyrażenie else rezultatGdyFalse
```

W Pythonie mamy do dyspozycji dwa rodzaje pętli: `while` i `for`.

```
while wyrażenie:
  instrukcje
```

alternatywnie

```
while wyrażenie:
  instrukcje1
else:
  instrukcje2
```

Instrukcje umieszczone w sekcji `else` będą wykonane wtedy, gdy pętla zostanie zakończona w normalny sposób, przez niespełnienie warunku (`wyrażenie` zwróci wartość równoważną `False`), a nie za pomocą instrukcji przerwania `break`. Notabene w Pythonie występuje także słowo kluczowe `continue` znane z C++ i Javy.

Także w pętli `for` mamy do wyboru dwie składnie, których semantyka jest podobna do tej obowiązującej w pętli `while`:

```
for zmienna in iterator:
  instrukcje1
else:
  instrukcje2
```

Przeważnie iteratorem jest: łańcuch, krotka, lista, zbiór, plik, słownik (przez użycie metod `iteritems()`, `iterkeys()`, `itervalues()`). Może nim być także dowolna klasa, o ile zdefiniujemy metodę `__iter__()` bądź `__getitem__()`. Funkcja również może być iteratorem; stosowny przykład zostanie przedstawiony w dalszej części podrozdziału. Generator uzyskamy także wtedy, gdy we wspomnianej wcześniej metodzie tworzenia list, nawiasy kwadratowe zastąpimy nawiasami zwykłymi, na przykład:

```
>>> for para in ((x,y) for x in xrange(2) for y in xrange(3)):
...   print para,
...
(0, 0) (0, 1) (0, 2) (1, 0) (1, 1) (1, 2)
```

W Pythonie wyróżniamy trzy rodzaje funkcji: zwykłe, anonimowe i metody. W tym punkcie opiszemy tylko te dwa pierwsze rodzaje, natomiast metody zostaną opisane w podrozdziale dotyczącym obiektowości.

Zakres działania funkcji może być globalny lub lokalny. Ogólnie mówiąc, zakres globalny oznacza, że funkcja jest dostępna w pliku,

w którym została zdefiniowana i w dowolnym pliku, który go zaimportował. Z lokalnym działaniem funkcji mamy do czynienia wtedy, gdy funkcja ta została zdefiniowana wewnątrz innego zakresu (np. wewnątrz innej funkcji).

Funkcje definiuje się za pomocą instrukcji `def`, według schematu:

```
def nazwaFunkcji(parametry):
  ciałoFunkcji
```

Na przykład:

```
def powitanie():
  print "Witam na zajęciach"
```

Nazwa funkcji jest referencją do obiektu i jako taka może być przypisana do innej zmiennej lub składowana w dowolnej strukturze danych:

```
f = powitanie
f()            # Zobaczymy napis "Witam na zajęciach"
```

Parametry są zawsze przekazywane przez referencję. Dopuszczalne jest także określanie wartości domyślnej (co znane jest np. z języka C++) oraz przekazywanie zmiennej liczby parametrów (znak $*$ przed parametrem), na przykład:

```
>>> def maxLiczba(*a):
...   return max([i for i in a if isinstance(i, int)])
...
>>> maxLiczba(3, 'a', 4)
4
>>> maxLiczba(1, 'bcd')
1
```

Przy przekazywaniu parametrów można wykorzystywać słowniki (znaki $**$ przed parametrem). Daje to swobodę w określaniu nazwanych parametrów:

```
>>> def opis(imie, **terminy):
...   print imie, "pracuje w sali",
...   for (dzien, sala) in terminy.iteritems():
...     print sala, "("+dzien+")",
...
>>> opis("Adam", pon=3, pt=2)
Adam pracuje w sali 3 (pon) 2 (pt)
>>> opis("Ewa", wt=2, czw=7, pt=3)
Ewa pracuje w sali 2 (wt) 7 (czw) 3 (pt)
```

Utworzenie iteratora z funkcji jest bardzo proste. Wystarczy zamienić instrukcję `return` zwracającą wynik na instrukcję `yield`. Na przykład, generator nieskończonego ciągu Fibonacciego można zdefiniować następująco:

```
def fib():
```

```
  n1 = 1
  yield 1
  n2 = 1
  while True:
    yield n2
    x = n1 + n2
    n1 = n2
    n2 = x
```

W większości zastosowań potrzebujemy skończonej liczby wygenerowanych elementów. Na przykład, generator wszystkich podzbiorów podanego zbioru typu `set` można zdefiniować rekurencyjnie:

```
def podzb(z):
  if len(z) == 0:
    yield z
  else:
    el = z.pop()
    for p in podzb(z):
      yield p
      yield set([el]) | p
```

Funkcja anonimowa (bez nazwy), zwana także funkcją lambda, tworzona jest za pomocą słowa kluczowego `lambda`. Na przykład:

```
kwadrat = lambda x: x*x
```

Nie może ona zawierać instrukcji warunkowych (ale może zawierać wyrażenie warunkowe `a if b else c`), pętli, ani instrukcji `return`. Zwracana jest przez nią wartość wyrażenia umieszczonego po dwukropku. Funkcje te nierzadko znajdują zastosowanie w połączeniu z funkcją `map` lub `reduce`:

```
>>> tab = ['ab', 'cde', 'fg']
>>> reduce(lambda x,y: x+y, tab)
'abcdefg'
```

Obsługę wyjątków w Pythonie realizuje się bardzo podobnie jak w języku Java. Składnia jest następująca:

```
try:
  instrukcje
except wyjątki:
  obsługa_wyjątków
else:
  sekcja_else
finally:
  sekcja_końcowa
```

Części `else` oraz `finally` można pominąć. Jest wiele sposobów przechwytywania wyjątków. Wybór właściwego zależy od tego, czy przechwytujemy wyjątek konkretnego typu, ale bez dodatkowego obiektu; z dodatkowym obiektem; jeden z kilku wymienionych; czy wreszcie dowolny wyjątek:

```
except IndexError: pass
except ValueError, ob: pass
except (IOError, OSError), ob: pass
except: pass
```

Możliwe jest zgłoszenie wyjątku przez własny program przez użycie instrukcji `raise`. Poniższe przykłady pozwolą oswoić się ze sposobem obsługi wyjątków w Pythonie.

```
# coding: utf-8

zb = set(str(i) for i in xrange(5))
cyfra = raw_input("Podaj cyfrę: ")
try:
  zb.remove(cyfra)
except KeyError:
  print "Brak klucza w zbiorze!"
else:
  print "Klucz został usunięty."
```

W kolejnym przykładzie mamy do czynienia z utworzeniem własnego wyjątku:

```
import math, random

class UjemnaLiczba(Exception): pass

tab = [1,3,2,4,5]
random.seed()

def pierwiastek(x):
  if x < 0:
    raise UjemnaLiczba, x
  else:
    return math.sqrt(x)

try:
  print pierwiastek(random.randint(-2, 2))
  print tab[random.randint(0, 10)]
except (UjemnaLiczba, IndexError), ob:
  print ob
```

Programowanie obiektowe

Podrozdział ten rozpoczniemy od krótkiej charakterystyki programowania obiektowego w języku Python, nierzadko przywołując podobieństwo do programowania obiektowego w Javie i C++. W Pythonie każda metoda może być przedefiniowana w podklasie; w terminologii języka C++ powiedzielibyśmy, że każda metoda Pythona jest wirtualna. Nie jest dozwolone przeciążanie metod w obrębie jednej klasy, co nie jest jednak istotnym ograniczeniem

ze względu na różne sposoby przekazywania parametrów. Z podobieństw do C++ można jeszcze wymienić możliwość przeciążania operatorów i wielokrotne dziedziczenie (wyprowadzenie klasy z kilku klas podstawowych). Nie ma natomiast klas abstrakcyjnych (co nie przeszkadza w realizacji tej idei w języku Python) i destruktorów. Z podobieństw do Javy wymienić należy istnienie superklasy (wspólnej nadklasy wszystkich innych klas) i automatyczne usuwanie nieużywanych obiektów (ang. garbage collection).

Klasę definiujemy korzystając z prostej składni:

```
class nazwaKlasy(klasa_bazowa):
  ciało_klasy
```

W ciele klasy można definiować metody za pomocą instrukcji `def`. Klasa może też być „pusta", taka jak klasa `UjemnaLiczba` zdefiniowana na końcu poprzedniego podrozdziału. Metody lub atrybuty (pola) klasy, których nazwy rozpoczynają się dwoma znakami podkreślenia są prywatne, wszystkie inne są publiczne[31]. Oprócz metod i atrybutów, klasa może mieć właściwości, które zostaną opisane w dalszej części podrozdziału.

[31] W rzeczywistości nawet do składowych prywatnych można odwoływać się spoza klasy za pomocą nazwy składającej się ze znaku podkreślenia, następującej po nim nazwie klasy, a dalej dwóch znaków podkreślenia i nazwy składowej prywatnej.

Tworzenie obiektów

Sposób tworzenia obiektu klasy bardzo przypomina ten znany z Javy; różnica wynika z braku deklaracji typu zmiennych w Pythonie:

```
ob = MojaKlasa(arg)
```

Rolę konstruktora pełni metoda o nazwie `__init__()`. Jeśli nie ma potrzeby inicjowania klasy argumentami, to metody `__init__()` nie trzeba definiować. Rozważmy następujący przykład:

```
class Wielomian(object):
  def __init__(self, wzor):
    self.__wzor = wzor
  def podajWzor(self):
    return self.__wzor
```

Pierwszym parametrem każdej (niestatycznej) metody musi być `self`, będący odpowiednikiem znanej z C++ i Javy zmiennej `this`, referencji do obiektu, dla którego wywołuje się składową. Metody z nazwami rozpoczynającymi i jednocześnie kończącymi się dwoma znakami podkreślenia określane są mianem metod specjalnych. Metody te służą do przystosowania klasy do możliwości wykorzystania jej obiektów zgodnie ze składnią obowiązującą obiekty wbudowanych klas, co pokażemy dalej.

Do składowych obiektu odwołujemy się — jak w większości obiektowych języków programowania — za pomocą kropki:

```
>>> p1 = Wielomian("2*x**4 + 7*x - 2")
>>> p1.podajWzor()
'2*x**4 + 7*x - 2'
```

Klasa `Wielomian` ma jeden atrybut. Został on zapisany z dwoma znakami podkreślenia na początku, więc jest prywatny i odwołanie się do niego spoza klasy jest niedopuszczalne:

```
>>> p1.__wzor
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Wielomian' object has no attribute '__wzor'
```

Atrybuty klasy są „deklarowane" i inicjowane w dowolnych metodach, po prostu przez nadanie im wartości. Co więcej, w Pythonie dopuszcza się dodanie nowego atrybutu do utworzonego już obiektu, na przykład:

```
>>> p1.nazwa_zmiennej = 'x'
```

Właściwości i metody specjalne

Rozważmy jeszcze raz klasę reprezentującą wielomian:

```
class Wielomian(object):
  def __init__(self, wzor, zmienna):
    self.__wzor = wzor
    self.__zmienna = zmienna
  def podajWzor(self):
    return self.__wzor
  def podajZmienna(self):
    return self.__zmienna
  def nowaZmienna(self, nowa):
    self.__wzor = self.__wzor.replace(self.__zmienna, nowa)
    self.__zmienna = nowa
```

Obiekt tej klasy można modyfikować w następujący sposób:

```
>>> w1 = Wielomian("-2*x**2 + 3*x + 1", "x")
>>> w1.podajWzor()
'-2*x**2 + 3*x + 1'
>>> w1.podajZmienna()
'x'
>>> w1.nowaZmienna("y")
>>> w1.podajWzor()
'-2*y**2 + 3*y + 1'
```

Okazuje się, że korzystanie z obiektów klasy `Wielomian` można uprościć przez dodanie właściwości do klasy. Odpowiada za to funkcja `property()`. Dzięki temu za pomocą jednej nazwy realizujemy jednocześnie ustalanie i pobieranie wartości atrybutu:

```
class Wielomian(object):
  def __init__(self, wzor, zmienna):
    self.__wzor = wzor
    self.__zmienna = zmienna
  def podajWzor(self):
```

```
    return self.__wzor
  def podajZmienna(self):
    return self.__zmienna
  def nowaZmienna(self, nowa):
    self.__wzor = self.__wzor.replace(self.__zmienna, nowa)
    self.__zmienna = nowa
  zmienna = property(fget=podajZmienna, fset=nowaZmienna)
```

Teraz dialog z interpreterem Pythona może przebiegać z użyciem właściwości `zmienna`:

```
>>> w1 = Wielomian("-2*x**2 + 3*x + 1", "x")
>>> w1.podajWzor()
'-2*x**2 + 3*x + 1'
>>> w1.zmienna
'x'
>>> w1.zmienna = "y"
>>> w1.podajWzor()
'-2*y**2 + 3*y + 1'
```

Nie ulega żadnej wątpliwości, że wykorzystanie koncepcji właściwości przyczynia się do zwiększenia czytelności kodu.

Tabela 6: Podstawowe metody specjalne

Nazwa	**Opis**	**Składnia**
`__init__(self, arg)`	Inicjuje nowo tworzony obiekt	`x = X()`
`__call__(self, arg)`	Możliwość traktowania obiektu jako funktora. Argumenty nie są wymagane.	`x()`
`__eq__(self, ob)`	Zwraca `True`, jeśli `x` i `y` są równe	`x == y`
`__ne__(self, ob)`	Zwraca `True`, jeśli `x` i `y` są różne	`x != y`
`__le__(self, ob)`	Zwraca `True`, jeśli `x` jest mniejsze lub równe `y`	`x <= y`
`__lt__(self, ob)`	Zwraca `True`, jeśli `x` jest mniejsze od `y`	`x < y`
`__ge__(self, ob)`	Zwraca `True`, jeśli `x` jest większe lub równe `y`	`x >= y`
`__gt__(self, ob)`	Zwraca `True`, jeśli `x` jest większe od `y`	`x > y`
`__nonzero__(self)`	Zwraca `True`, gdy `x` jest wartością niezerową	`if x:`
`__str__(self)`	Zwraca łańcuchową reprezentację `x`	`print x`

Metody specjalne (niektóre z nich przedstawiono w tabeli 6) służą do przeciążania operatorów, ale także do umożliwienia stosowania obiektów tak, jakby były one wbudowane, na przykład jak kontenery. W tym drugim przypadku najczęściej definiowanymi metodami specjalnymi są te przedstawione w tabeli 7.

Załóżmy, że do klasy `Wielomian` dodano metodę specjalną `__call__()`:

```
def __call__(self, liczba):
```

```
  exec(self.__zmienna + "=" + str(liczba))
  return eval(self.__wzor)
```

Wówczas, dzięki przeciążeniu operatora wywołania funkcji, można w naturalny sposób zapisywać wyznaczanie wartości wielomianu w podanym punkcie:

```
>>> w = Wielomian("x**2 - 2*x + 1", "x")
>>> w(3)
4
```

Tabela 7: Wybrane metody specjalne używane w operacjach kontenerowych

Nazwa	Opis	Składnia
`__contains__(self, x)`	Zwraca `True`, jeśli `x` znajduje się w sekwencji `y` lub jeśli `x` jest kluczem w słowniku `y`. Metoda ta odpowiada również za sprawdzanie `not in`.	`x in y`
`__len__(self)`	Zwraca liczbę elementów w `y`	`len(y)`
`__getitem__(self, k)`	Zwraca `k`-ty element sekwencji `y` lub wartość w słowniku `y` skojarzoną z kluczem `k`	`y[k]`
`__setitem__(self, k, v)`	Nadaje wartość `v` `k`-temu elementowi sekwencji `y` lub elementowi w słowniku `y` skojarzonemu z kluczem `k`	`y[k] = v`
`__delitem__(self, k)`	Usuwa `k`-ty element sekwencji `y` lub klucz `k` wraz ze skojarzonym z nim obiektem w słowniku `y`	`del y[k]`
`__iter__(self)`	Zwraca iterator kontenera `y`	`for x in y:`

Składowe statyczne

Klasa jest typem, nie obiektem danych i każdy obiekt klasy ma swoją własną kopię atrybutów klasy. Jednak, niektóre typy implementuje się bardziej elegancko, jeśli wszystkie obiekty tego typu współdzielą pewne dane. Najlepiej byłoby, gdyby takie statyczne (dzielone) dane były zadeklarowane jako część klasy. Sposób realizacji tej idei jest podobny do tego stosowanego w C++ i Javie. Dana statyczna tworzona jest w ciele klasy, jednak poza zakresem jakiejkolwiek instrukcji `def`. Metoda statyczna natomiast nie posiada parametru `self` i opatrywana jest dekoratorem `@staticmethod`:

```
class Dana(object):
  li = 0
  ls = 0
  def __init__(self, x):
    self.x = x
    if isinstance(x, int):
```

```
      Dana.li += 1
    elif isinstance(x, str):
      Dana.ls += 1
  @staticmethod
  def ileOb():
    return Dana.li + Dana.ls
```

Oczywiście do składowych statycznych można się odwoływać bez tworzenia obiektu klasy `Dana`:

```
>>> Dana.ileOb()
0
>>> o1 = Dana(3)
>>> o2 = Dana("abc")
>>> o3 = Dana("d")
>>> Dana.ileOb(), Dana.li, Dana.ls
(3, 1, 2)
```

Dziedziczenie i polimorfizm

Dziedziczenie realizowane jest za pomocą naturalnej konstrukcji:

```
class Podklasa(Klasa_podstawowa):
  ciało_klasy
```

W przypadku dziedziczenia wielokrotnego używamy składni:

```
class Podklasa(Podstawowa1, Podstawowa2, Podstawowa3):
  ciało_klasy
```

Konstruktor klasy podstawowej może być wywołany na dwa sposoby: za pomocą wbudowanej funkcji `super()` lub bezpośrednio, jak w klasie `C` z poniższego przykładu:

```
class A(object):
  def __init__(self, a):
    self.a = a

class B(A):
  def __init__(self, a, b):
    super(B, self).__init__(a)
    self.b = b

class C(A):
  def __init__(self, a, c):
    A.__init__(self, a)
    self.c = c
```

W Pythonie pozwala się na przedefiniowanie w każdej klasie pochodnej funkcji zadeklarowanych w klasie podstawowej. Czyli każda metoda `m()` może mieć różne wersje dla różnych klas pochodnych i znalezienie właściwej metody dla każdego wywołania `m()` jest zadaniem interpretera (tzw. polimorfizm).

Chociaż w Pythonie nie ma możliwości deklarowania czystej funkcji wirtualnej[32] (jak w C++) ani interfejsów (jak w Javie), nic nie stoi na przeszkodzie, aby łatwo osiągnąć ten sam efekt:

[32] Uważny czytelnik zapamiętał zapewne, że autor pierwszego rozdziału tego skryptu nalega na stosowanie nazwy *funkcja abstrakcyjna*.

```
class Osoba(object):
  def identyfikator(self):
    raise NotImplementedError, "Osoba.identyfikator()"
  def kwalifikacje(self):
    raise NotImplementedError, "Osoba.kwalifikacje()"
```

Przykłady

Rozważmy wyrażenia regularne, w których nie używamy domknięcia Kleene'ego[33]. Zostają więc tylko operacje złożenia, które będziemy oznaczać za pomocą $*$, i alternatywy, którą będziemy oznaczać za pomocą `|`. Formalnie zdefiniujmy nasze wyrażenia regularne i zbiory słów przez nie reprezentowane w następujący sposób:

[33] Domknięciem Kleene'ego zbioru łańcuchów L nazywamy zbiór $\bigcup_{i=0}^{\infty} L^i$

1. każda litera a jest wyrażeniem regularnym reprezentującym zbiór $\{a\}$,
2. jeżeli r i s są wyrażeniami regularnymi reprezentującymi, odpowiednio, zbiory słów R i S, to $r|s$ i $r * s$ są wyrażeniami regularnymi reprezentującymi, odpowiednio, zbiory $R \cup S$ i RS.

Aby przy zapisywaniu wyrażeń regularnych można było opuścić wiele nawiasów, przyjmiemy, że $*$ ma wyższy priorytet niż `|` (w języku Python tak właśnie jest).

Zadanie polega na tym, aby skończone zbiory słów można było zapisywać za pomocą wyrażeń regularnych i iterować ich słowa:

```
>>> for slowo in a|b|c*a|a:
...   print slowo,
...
a ca b
```

Chociaż zadanie wydaje się trudne, można się z nim uporać bardzo szybko, przyjmując za podstawę wyrażeń regularnych wbudowany typ `set`. Operator `|` jest tam zdefiniowany dokładnie w taki sposób, jakiego oczekujemy. Wystarczy przeciążyć operator $*$ tak, aby generował on złożenie zbioru słów:

```
class Reg(set):
  def __mul__(self, r):
    return Reg(x+y for x in self for y in r)
```

Korzystanie z klasy `Reg` jest bardzo wygodne:

```
>>> a = Reg(["a"])
>>> b = Reg(["b"])
>>> for slowo in (a|b)*(a|b)*(a|b):
...   print slowo,
...
aba abb aaa aab bbb bba bab baa
```

W kolejnym przykładzie również będziemy mieli do czynienia ze światem języków formalnych. Załóżmy, że naszym zadaniem jest uzyskanie kilku tysięcy słów generowanych przez gramatykę bezkontekstową zawierającą jedną zmienną (tzw. symbol nieterminalny). Na przykład, palindromy generowane przez gramatykę:

$$S \to \epsilon \mid a \mid b \mid aSa \mid bSb$$

chcemy uzyskać za pomocą kodu:

```
epsilon = Stala('')
a = Stala('a')
b = Stala('b')
S = Zmienna()

S <<= epsilon
S <<= a
S <<= b
S <<= a, S, a
S <<= b, S, b

for (nr, slowo) in zip(xrange(100), S):
  print nr, slowo
```

Stałe i zmienna gramatyki mają cechę wspólną — są symbolami gramatyki. Stwórzmy zatem hierarchię klas:

```
class Symbol(object): ...
class Stala(Symbol): ...
class Zmienna(Symbol): ...
```

Ponieważ każdy symbol gramatyki można utożsamić ze zbiorem słów przez niego generowanym, atrybutem klasy `Symbol` będzie zmienna `slowa` typu `set`. Ze specyfikacji wymagań wynika, że w klasie `Zmienna` należy przeciążyć metody `__ilshift__()` (aby móc używać operatora `<<=`) i `__iter__()` (aby móc generować dowolną liczbę słów za pomocą pętli `for`). Zauważmy też, że po prawej stronie operatora `<<=` może znajdować się pojedynczy symbol lub krotka symboli. Działanie tego operatora będzie polegało na dopisywaniu kolejnych reguł do listy reguł produkcji określających sposób powstawania słów w implemenowanej gramatyce. Główna część obliczeniowa — czyli generowanie słów — zostanie zrealizowana w metodzie `__iter__()` i opiera się na następującej zasadzie. Jeśli dla symbolu a przez $a.s$ oznaczymy słowa, które ten symbol generuje, to dla gramatyki:

$$Z \to r_{11}r_{12}\ldots r_{1k_1} \mid \ldots \mid r_{n1}r_{n2}\ldots r_{nk_n}$$

zbiór słów $L(Z)$ przez nią generowany jest równy:

$$L(Z) = \bigcup_{i=1}^{n}(\cdots(((r_{i1}.s\ r_{i2}.s)r_{i3}.s)r_{i4}.s)\cdots r_{ik_i}.s).$$

W programie tę operację sumowania zbiorów słów powtarzamy wielokrotnie, bo zbiór `self.slowa` w każdej iteracji ma coraz więcej elementów.

```
class Symbol(object):
  def __init__(self, slowa):
    self.slowa = slowa
  def __mul__(self, s):
    return Symbol(x+y for x in self.slowa for y in s.slowa)

class Stala(Symbol):
  def __init__(self, s):
    super(Stala, self).__init__(set([s]))

class Zmienna(Symbol):
  def __init__(self):
    super(Zmienna, self).__init__(set())
    self.__reguly = []
    self.__pokazane = set()
  def __ilshift__(self, r):
    if isinstance(r, tuple):
      self.__reguly.append(r)
    else:
      self.__reguly.append((r,))
    return self
  def __iter__(self):
    zb = set()
    while True:
      zb.clear()
      for r in self.__reguly:
        zb.update(reduce(lambda x,y: x*y, r).slowa)
      self.slowa.update(zb)
      for slowo in self.slowa:
        if slowo not in self.__pokazane:
          self.__pokazane.add(slowo)
          yield slowo
```

Prywatny atrybut `pokazane` został umieszczony w klasie `Zmienna` po to, aby przez metodę `__iter__()` to samo słowo nie zostało zwrócone dwa razy.

Podsumowanie

W rozdziale tym, na wstępie przedstawiono ogólną informację o języku Python. Następnie szczegółowo opisano, w jaki sposób definiuje się klasy i tworzy obiekty tych klas — czyli fundamenty programowania zorientowanego obiektowo. Szeregiem przykładów umieszczonych w dalszej części rozdziału, zilustrowano, że w języku Python równie łatwo realizuje się inne zasadnicze cechy programowania obiektowego: dziedziczenie i polimorfizm, a także

techniki zaawansowane jak składowe statyczne lub przeciążanie operatorów. Pomimo pewnych różnic w stosunku do innych, „wzorcowych” języków takich jak C++, SmallTalk i Java, język Python należy zaliczyć do grupy podstawowych języków programowania obiektowego.

Warto wspomnieć, co — poza prostą syntaktyką — utrwaliło popularność języka Python wśród programistów. Otóż niewątpliwie jest to duża liczba gotowych modułów, dzięki którym spektrum zastosowań tego języka jest bardzo szerokie. Na oficjalnej stronie Pythona (`www.python.org`) można znaleźć moduły dotyczące między innymi: kryptografii, baz danych, systemów informacji geograficznej, przetwarzania obrazów, komunikacji sieciowej, wielowątkowości, formatu XML, a nawet wspomagania projekowania gier. Dla potrzeb inżynierów i naukowców opracowano biblioteki NumPy i SciPy (obie są dostępne na stronie `www.scipy.org`). Zawierają one funkcje do realizowania zaawansowanych zadań ze statystyki, algebry, optymalizacji numerycznej, przetwarzania sygnałów oraz operacji na macierzach. Dostępna jest tam także implementacja metod numerycznego całkowania i różniczkowania. Osobom, które rozważane problemy formułują jako zagadnienie programowania liniowego lub całkowitoliczbowego, pomocną okaże się biblioteka PuLP[34], stanowiąca pomost między Pythonem a takimi pakietami oprogramowania do rozwiązywania problemów programowania liniowego o dużej skali trudności, jak GLPK, CPLEX, Gurobi, CoinMP i in.

[34] `https://www.coin-or.org/PuLP/`

Zadania

Zad. 1. Zaimplementuj klasę `Tablice` tak, aby można było z niej korzystać w następujący sposób:

```
>>> t = Tablice(3)
>>> for i in t:
...   print i
...   if len(i) >= 4:
...     break
...
[3]
[3, 4]
[3, 4, 5]
[3, 4, 5, 6]
>>> t.pocz = 9
>>> for i in t:
...   print i
...   if len(i) >= 2:
...     break
...
[9]
[9, 10]
```

Zad. 2. Permutację zbioru liczb $\{1, 2, \ldots, n\}$ można przedstawiać w różny sposób. Wygodną strukturą danych jest krotka n różnych liczb, np. krotkę $(2, 1, 3)$ odczytujemy jako: z 1 do 2, z 2 do 1, z 3 do 3. Permutację można także przestawiać jako iloczyn cykli, np. $(2, 3, 1, 5, 4)$ rozkłada się na dwa cykle $(1, 2, 3)(4, 5)$. Mówimy, że permutacja π działa na liczbie a w wyniku czego dostajemy liczbę b, co zapisujemy $\pi(a) = b$, jeżeli w permutacji π występuje przejście z a do b.

Zaimplementuj klasę `Permutacja`, inicjowaną ciągiem liczb od 1 do n, dokonującą jej rozkładu na cykle oraz zwracającą wynik jej działania na podanej liczbie:

```
>>> p = Permutacja(1,4,5,2,3)
>>> p.naCykle()
[(1,), (2, 4), (3, 5)]
>>> p(3)
5
```

Zad. 3. Wiadomo, że liczbę zespoloną $a + bi$ można sprowadzić do postaci trygonometrycznej

$$a + bi = r(\cos\phi + i\sin\phi),$$

gdzie liczba dodatnia r jest modułem ($r = \sqrt{a^2 + b^2}$), a ϕ argumentem danej liczby zespolonej. W standardowej klasie `complex` nie przeciążono operatora mniejszości. Z klasy `complex` wyprowadź klasę `Zespolona`, w której będzie przeciążony operator mniejszości na zasadzie porównywania argumentów (kątów ϕ). Przykład jej użycia pokazano poniżej.

```
>>> Zespolona(2+1j) < Zespolona(1+2j)
True
```

Zad. 4. Zaimplementuj klasę do obsługi zbiorów z powtórzeniami, w których każdy element występuje określoną liczbę razy. Każdy zbiór z powtórzeniami A możemy reprezentować przez funkcję r_A, która każdemu elementowi x przyporządkowuje nieujemną liczbę całkowitą $r_A(x)$ zwaną *współczynnikiem repetycji* elementu x w zbiorze z powtórzeniami A. Działania $A \cup B$, $A \cap B$, $A - B$ definiujemy na zbiorach z powtórzeniami następująco:

$$\begin{aligned} r_{A\cup B}(x) &= \max(r_A(x), r_B(x)), \\ r_{A\cap B}(x) &= \min(r_A(x), r_B(x)), \\ r_{A-B}(x) &= \max(r_A(x) - r_B(x), 0). \end{aligned}$$

Zad. 5. Zaimplementuj klasę `Graf` — reprezentującą graf skierowany — tak, aby można było z niej korzystać jak z iteratora po wszystkich podgrafach danego grafu:

```
g = Graf(set([1,2,3]), set([(1,2), (2,3), (3,1)]))
for podgraf in g:
  print podgraf
```

W wyniku wykonania powyższego kodu powinniśmy uzyskać podgrafy:

```
({1}, {})
({2}, {})
({3}, {})
({1,2}, {})
({1,2}, {(1,2)})
({1,3}, {})
({1,3}, {(3,1)})
({2,3}, {})
({2,3}, {(2,3)})
({1,2,3}, {})
({1,2,3}, {(1,2)})
({1,2,3}, {(2,3)})
({1,2,3}, {(3,1)})
({1,2,3}, {(1,2), (2,3)})
({1,2,3}, {(1,2), (3,1)})
({1,2,3}, {(2,3), (3,1)})
({1,2,3}, {(1,2), (2,3), (3,1)})
```

Zad. 6. Zaimplementuj klasy `Zmienna`, `UkladRownan` oraz przypuszczalnie inne dodatkowe klasy, aby możliwe było wykonywanie następującego przykładowego kodu:

```
x = Zmienna('x', frozenset(range(1,10)))
y = Zmienna('y', frozenset([2,3,7]))

ur = UkladRownan(x, y)
ur += 2*x - 7*y == 3
ur += -x + 2*y == -9

for odp in ur.rozwiaz():
  print odp
```

Zakładamy, że:

- zmiennych może być dowolnie dużo,
- po lewej stronie każdego równania jest dowolne wyrażenie arytmetyczne złożone ze zmiennych (obiektów klasy `Zmienna`), stałych całkowitych, operatorów mnożenia, dodawania i odejmowania,
- po prawej stronie każdego równania jest liczba całkowita,
- każda zmienna może przyjmować jedynie te wartości całkowite, które określono w drugim parametrze konstruktora klasy `Zmienna`,

- metoda `rozwiaz()` zwraca listę dopuszczalnych rozwiązań podanego układu równań.

Zad. 7. Zaimplementuj klasę `DAS` reprezentującą deterministyczny automat skończony. Zdefiniuj operatory `|`, `&` i `~` tak, aby zwracały one, odpowiednio, automat reprezentujący sumę języków akceptowanych przez automaty będące argumentami, automat reprezentujący przecięcie języków i automat reprezentujący wszystkie słowa nieakceptowane przez automat będący argumentem wywołania (nad takim samym alfabetem). Dodatkowo zdefiniuj cztery metody:

1. sprawdzającą czy podane słowo jest akceptowane przez automat,
2. zwracającą minimalny DAS (mający najmniejszą możliwą liczbę stanów) równoważny danemu automatowi,
3. `__iter__()` do generowania wszystkich słów akceptowanych przez automat (w rozsądnym czasie),
4. `__str__()` do łańcuchowego przedstawiania automatu.

Zad. 8. Zaimplementuj kolejkę priorytetową w taki sposób, aby dało się jej używać w instrukcjach o postaci:

```
>>> k = Kolejka()
>>> k << 2 << 5 << 1
>>> print k.getmax()
5
>>> k << 3
>>> print k.getmax(), k.getmax(), k.getmax()
3 2 1
>>> print len(k)
0
```

Zad. 9. Zaimplementuj klasę `Relacja` reprezentującą tabelę w relacyjnej bazie danych. Klasa ta powinna udostępniać wszystkie operacje, które zostały zaprezentowane w poniższym przykładzie:

```
>>> osoby = Relacja(nazwisko=str, wiek=int)
>>> osoby.dopisz(nazwisko='Nowak', wiek=19)
>>> osoby.dopisz(nazwisko='Nowak', wiek=20)
>>> osoby.dopisz(nazwisko='Kowalski', wiek=20)
>>> for k in osoby:
...   print k
...
( nazwisko='Nowak'  wiek=19 )
( nazwisko='Nowak'  wiek=20 )
( nazwisko='Kowalski'  wiek=20 )
>>> odp1 = osoby("wiek == 20")
>>> for k in odp1:
...   print k
```

```
...
( nazwisko='Nowak'  wiek=20 )
( nazwisko='Kowalski'  wiek=20 )
>>> odp2 = osoby("nazwisko == 'Nowak' and wiek < 20")
>>> for k in odp2:
...   print k
...
( nazwisko='Nowak'  wiek=19 )
```

Zadanie to rozwiąż w taki sposób, aby zmienne `odp1` i `odp2` również były obiektami klasy `Relacja`.

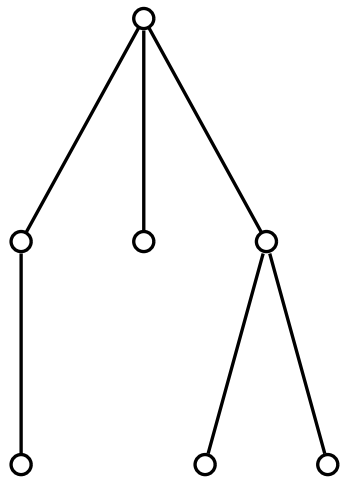
Zad. 10. Niech w będzie słowem utworzonym wyłącznie z liter x i y. Powiemy, że w jest słowem Dycka, jeśli w jest słowem pustym lub słowem $xuyv$, gdzie u i v są pewnymi słowami Dycka. Oznaczając przez D język słów Dycka, możemy napisać:

$$D = \epsilon + xDyD.$$

Niech $s = xt$, gdzie t jest dowolnym prefiksem słowa Dycka. Istnieje wzajemnie jednoznaczne odwzorowanie pomiędzy drzewami z n wierzchołkami i słowami s z n literami x i $n - 1$ literami y. Oto algorytm tworzący drzewo na podstawie słowa s z n literami x i $n - 1$ literami y:

1. Zainicjuj pusty stos.
2. Czytaj litery słowa s od lewej do prawej strony. Za każdym razem, kiedy napotkasz literę x, a za nią k ($k \geq 0$) liter y, ściągnij ze stosu k drzew, $T_1, \ldots, T_k$ i połóż na stosie drzewo utworzone z nowego korzenia i dołączonych do niego synów $T_1, \ldots, T_k$.
3. Po przeczytaniu całego słowa s na stosie pozostanie jedno (unikatowe) drzewo $g(s)$ odpowiadające słowu s.

Na przykład słowu $s = xxyxxxxyyxyyy$ odpowiada drzewo przedstawione na rys. 34. Zaimplementuj powyższy algorytm, przyjmując następującą definicję drzewa:



Rysunek 34: Drzewo utworzone na podstawie słowa s

```
class Drzewo(object):
  def __init__(self):
    self.__synowie = []
  def dodajSyna(self, syn):
    self.__synowie.append(syn)
  def drukuj(self, poziom=0):
    print poziom*" ", "o"
    for syn in self.__synowie:
      syn.drukuj(poziom+1)
```

oraz listę (obiekt klasy `list`) z metodami `append` i `pop` jako stos.

Bibliografia

D. Alur, J. Crupi, and D. Malks. *J2EE. Wzorce projektowe*. Helion, 2004.

Brad Dayley. *Python: Rozmówki*. Helion, 2007.

Informacje dostępne online. The Development of the C Language by Dennis Ritchie. *http://cm.bell-labs.com/cm/cs/who/dmr/chist.html*, 2009a.

Informacje dostępne online. Bjarne Stroustrup's homepage. *http://www.research.att.com/~bs/*, 2009b.

B. Eckel. *Thinking in Java*. Helion, 2006.

I. Graham, A. O'Callaghan, and A. Wills. *Metody obiektowe w teorii i w praktyce*. WNT, 2004.

C.S. Horstmann and G. Cornell. *Java. Techniki zaawansowane*. Helion, 2009.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

Jeremy Gibbons i Oege de Moor (ed.). *The Fun of Programming*. Palgrave Macmillan, 2003.

E. Koffman and P. Wolfgang. *Struktury danych i techniki obiektowe na przykładzie Javy 5.0*. Willey/Helion, 2006.

Hans Petter Langtangen. *A Primer on Scientific Programming with Python*. Springer, 2009.

Lajoie J. Lippman S.B. *Podstawy języka C++*. Wydawnictwa Naukowo-Techniczne, Warszawa, Polska, 2003.

Mark Lutz and David Asher. *Python: Wprowadzenie*. Helion, 2002.

David Mertz. *Text Processing in Python*. Addison Wesley, 2003.

Josuttis N.M. *C++. Biblioteka standardowa. Podręcznik programisty*. Helion, Gliwice, Polska, 2003.

Peter C. Norton, Alex Samuel, Dave Aitel, Eric Foster-Johnson, Leonard Richardson, Jason Diamond, Aleatha Parker, and Michael Roberts. *Beginning Python*. Wrox, 2005.

Plauger P.J. *Biblioteka standardowa C++*. Wydawnictwa Naukowo-Techniczne, Warszawa, Polska, 1997.

Fethi Rabhi and Guy Lapalme. *Algorithms: a functional programming approach*. Addison-Wesley, 1999.

Meyers S. *STL w praktyce. 50 sposobów efektywnego wykorzystania*. Helion, Gliwice, Polska, 2004.

Prata S. *Szkoła programowania. Język C++*. Helion, Gliwice, Polska, 2006.

H. Schildt. *Java. Kompendium programisty*. Helion, 2005.

Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999.