Informatyka Inżynierska – Kierunek Zamawiany Uniwersytetu Śląskiego w Katowicach
Priorytet IV – Szkolnictwo wyższe i nauka, Poddziałanie 4.1.2 Programu Operacyjnego Kapitał Ludzki

Uniwersytet Śląski w Katowicach, ul. Bankowa 12, 40-007 Katowice, http://www.us.edu.pl

Arkadiusz Sacewicz

Java

Materiały dydaktyczne, e-learningowe do modułu:

Programowanie Obiektowe.

Uniwersytet Śląski 2013

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Autor serdecznie dziękuję wszystkim osobom, które przyczyniły się do powstania książki w obecnej formie,

w szczególności:

Urszuli i Mariuszowi Boryczkom,

Wojciechowi Wieczorkowi.

Wstęp

Java jest obiektowym językiem programowania. Powstała jak uniwersalny język programowania urządzeń przemysłowych i domowych. Rozwój języka oraz jego uniwersalność spowodowały, że obecnie jest z powodzeniem stosowana, jako język programowania aplikacji biznesowych. Do uruchomienia aplikacji napisanej w Javie wystarczy dowolny komputer z zainstalowaną maszyną Javy. Jesteśmy niezależni od sprzętu oraz systemu operacyjnego użytkownika. Programista Javy może korzystać z bogatej biblioteki standardowych klas oraz różnych bibliotek/środowisk (frameworków) oferowanych przez firmy trzecie i społeczności.

Zmienne

Zmienne umożliwiają przechowywanie danych różnego typu (o typach powiemy sobie w następnym punkcie). Aby móc korzystać ze zmiennej wymagane są dwie czynności:

- deklaracja;
- inicjalizacja.

Deklaracja – określa typ danych przechowywanych przez zmienną oraz nazwę (identyfikator) zmiennej. Zapis formalny:

```
identyfikatorTypu identyfikatorZmiennej;
```

Inicjalizacja – umożliwia nadanie wartości zmiennej. Zapis formalny:

```
identyfikatorZmiennej = wyrażenie;
```

np.:

```
int licznik; // deklaracja zmiennej;
licznik = 1; //inicjalizacja zmiennej;
String napis; // deklaracja zmiennej;
napis = "Napis ćwiczebny"; //inicjalizacja zmiennej.
```

Często łączymy te dwie czynności jeden zapis będący równocześnie deklaracją oraz inicjalizację. np.

```
int licznik = 1;
String napis = "Napis ćwiczebny"
```

Identyfikatorem zmiennej może być dowolny ciąg znaków rozpoczynający się literą. Każda zmienna jest widoczna wewnątrz tego bloku, wewnątrz którego została zadeklarowana.

Możemy zadeklarować zmienną w klasie, poza metodami, będzie ona widoczna dla wszystkich metod tej klasy, np.:

```
class MojaKlasa{
      void mojaMetoda1(){
            …
            mojaZmienna=15; // można odwołać się do zmiennej
            …
      }
      double mojaMetoda2(){
            return mojaZmienna; // można odwołać się do zmiennej
      }
      double mojaZmienna;
}
```

Możemy zadeklarować zmienną wewnątrz metody, wtedy zmienna będzie widoczna tylko w tej metodzie, np.:

```
class MojaKlasa{
      void mojaMetoda1(){
            …
            double mojaZmienna;
            …
            mojaZmienna=15; // można odwołać się do zmiennej
            …
      }
      double mojaMetoda2(){
            return mojaZmienna; // błąd,
                                // nie można odwołać się do zmiennej
      }
}
```

Ponieważ **mojaZmienna** jest zadeklarowana wewnątrz metody **mojaMetoda1** nie można odwoływać się do niej z metody **mojaMetoda2**. Możemy powiedzieć, że z chwilą zamknięcia klamry kończącej metodę **mojaMetoda1**, ta zmienna „znika" z pamięci tak jakby nigdy nie istniała.

Możemy deklarować zmienne nawet wewnątrz instrukcji (bloku), wtedy taka zmienna jest widoczna tylko wewnątrz tej instrukcji, np.:

```
class MojaKlasa{
      void mojaMetoda1(){
            if(jakisWarunek){
            double mojaZmienna;
            …
            mojaZmienna=15; // można odwołać się do zmiennej
            …
            }
            mojaZmienna+=10; // błąd,
                             // nie można odwołać się do zmiennej
      }
      double mojaMetoda2(){
            return mojaZmienna; // błąd,
                                // nie można odwołać się do zmiennej
      }
}
```

Podobnie jak było ze zmiennymi deklarowanymi wewnątrz funkcji, zmienna, która została zadeklarowana wewnątrz instrukcji jest usuwana z pamięci z chwilą zamknięcia klamry kończącej tę instrukcję.

W następnym rozdziale zostaną omówione typy, które możemy używać dla zmiennych.

Typy proste

Java posiada 4 typy dla liczb całkowitych, różnią się one zajętością pamięci.

```
byte   - 1 bajt   - zakres od -128 do 127
short  - 2 bajty  - zakres od -32 768 do 32 767
int    - 4 bajty  - zakres od -2 147 483 648 do 2 147 483 647
long   - 8 bajtów - zakres od -2^63 do (2^63)-1
                    (posiadają przyrostek L, lub l)
```

Zazwyczaj dla liczb całkowitych posługujemy się typem **`int`**.

Standardowo Java nie kontroluje zakresu zmiennych, więc przekraczając zakres otrzymujemy wartości o przeciwnym znaku. Java nie posiada też typów całkowitych bez znaku.

Liczby rzeczywiste reprezentują dwa typy różniące się ilością zajmowanego miejsca, precyzją oraz zakresem wartości:

```
float  - 4 bajty  - max ok 6-7 liczb po przecinku
                  (posiadają przyrostek F, lub f)
double - 8 bajtów - max ok 15 cyfr po przecinku
                  (posiadają przyrostek D, lub d)
```

Podając wartości liczb w kodzie programu część całkowitą od ułamkowej oddzielamy kropką. Ze względu na to, że w systemie dwójkowym nie da się przedstawić wszystkich liczb, do obliczeń wymagających precyzji części ułamkowej należy używać klasy **`BigDecimal`**.

Typ char służy do reprezentacji pojedynczych znaków kodu **`Unicode`**. Wartości można podawać umieszczając je w pojedynczym cudzysłowie, dziesiętnie albo szesnastkowo, np. `,x'`, lub `\u0123`.

Wprowadzając znaki specjalne, poprzedzamy je znakiem backslash \:

```
\n - nowa linia
\r - powrót karetki
\t - tab
\" - cudzysłów
\' - apostrof
\\ - backslash
```

Wartości logiczne reprezentuje typ **`boolean`**:

```
true - prawda
false - fałsz
```

Podstawowe instrukcje

Instrukcja przypisania

Instrukcja przypisania, to instrukcja postaci:

```
zmienna = wyrazenie;
```

Nadaje ona zmiennej **zmienna** wartość określoną wyrażeniem **wyrazenie**. Wyrażenie jest dowolnym wyrażeniem matematycznym, logicznym itp. posiadającym wartość typu zgodnego (sensie przypisania) ze zmienną. np.:

```
int a,b,c,x,y;
double z;
//…

a = 10;
b = x / y;
c = z / a; //źle
```

W ostatniej linijce przykładu mamy błąd. Z jest typu double, więc wartość wyrażenia też jest typu rzeczywistego, a próbujemy to przypisać do zmiennej całkowitej. Jest to możliwe, ale musimy zaokrąglić wartość do całkowitej (tracimy precyzję wyniku).

Instrukcja warunkowa

Instrukcja warunkowa ma postać:

```
if (warunek) instrukcjaWewnetrzna;

if (warunek) instrukcjaWewnetrzna1;
else instrukcjaWewnetrzna2;
```

Umieszczony w nawiasach **warunek** jest dowolnym wyrażeniem posiadającym wartość logiczną (prawda lub fałsz). W pierwszym wariancie instrukcji **if instrukcjaWewnetrzna** wykona się tylko wtedy, gdy warunek będzie miał wartość logiczną **true**. W drugim wariancie, jeżeli warunek będzie miał wartość **true** to wykona się **instrukcjaWewnętrzna1**, a jeżeli **false** to **instrukcjaWewnętrzna2**.

Instrukcja wyboru

Instrukcja wyboru umożliwia wykonanie jednej z kilku instrukcji w zależności od wartości zmiennej użytej do sterowania instrukcją. Formalnie instrukcja ma następującą składnię:

```
switch (zmiennaStrujaca) {
     case wartosc1: instrukcjaWewnetrzna1;
     break;
     case wartosc2: instrukcjaWewnetrzna2;
     case wartosc3: istrukcjaWewnetrzna3;
}
```

Zastosowana **zmiennaSterująca** musi być typu: **char**, **byte**, **short**, **int**, **Character**, **Byte**, **Short**, **Integer**, **String**, lub **enum**. Jeżeli **zmiennaSrerujaca** przyjmie wartość: **wartosc1** to zostanie wykonana instrukcja **instrukcjaWewnetrzna1**. Jeżeli **zmiennaSrerujaca** przyjmie wartość: **wartosc2** to zostanie wykonana instrukcja **instrukcjaWewnetrzna2** oraz **instrukcjaWewnetrzna3**. Jest to spowodowane brakiem słowa kluczowego **break** po **instrukcjaWewnetrzna2**.

Pętle

Pętle umożliwiają wielokrotne wykonywanie innych instrukcji w zależności od wartości logicznej zastosowanego w nich warunku. W języku Java mamy do dyspozycji trzy instrukcje pętli: **do**, **while** oraz **for**.

1. Pętla **do**

Pętla **do** wykonuje przynajmniej jeden raz instrukcję zawartą wewnątrz niej, ponieważ warunek jest sprawdzany na końcu pętli. Składnia formalna jest następująca:

```
do instrukcja; while (warunek);
```

2. Pętla **while**

Instrukcja zawarta wewnątrz pętli **while** może nie zostać wykonana ani razu, ponieważ w tej pętli warunek jest sprawdzany na samym początku, jeszcze przed jej wykonaniem. Składnia formalna jest następująca:

```
while (warunek) instrukcja;
```

3. Petla **`for`**

Pętla **`for`** jest stosowana w większości przypadków tam, gdzie zakładamy z góry określoną liczbę wykonań pętli, co nie oznacza, że nie da się jej stosować w innych. Wynika to z jej zwartego zapisu, który przedstawia się następująco:

```
for(inicjalizacjaZmiennej; warunek; sterowanieZmienną) instrukcja;
```

Dodatkowo możemy sterować wykonywaniem pętli za pomocą instrukcji **`break`** i **`continue`**.

Wprowadzenie do programowania obiektowego

W języku Java można w pełni korzystać z zalet programowania obiektowego. Programista tworząc swoją klasę może zawrzeć w niej wszystkie funkcje związane z jakimś elementem (obiektem) programu. Klasa zawiera zarówno dane związane z obiektem, jak również metody umożliwiające manipulację tymi danymi. Zalety programowania obiektowego ujawniają się głównie przy programowaniu zespołowym oraz przy tworzeniu dużych projektów. W zespole programistów, gdy każdy odpowiada za spójność i funkcjonalność swojej klasy, można wydajnie tworzyć aplikacje bez znajomości kodu stworzonego przez innych programistów. Porozumienie jest konieczne tylko na poziomie „interfejsu" klasy. Zespół musi mieć uzgodnioną funkcjonalność poszczególnych obiektów:

- dane, którymi zarządzają obiekty;
- metody (z parametrami), które udostępniają obiekty.

Nie jest potrzebna wiedza, jak wygląda implementacja danej klasy (jej wnętrze), a nawet więcej, podczas rozwoju projektu ta implementacja może się zmieniać. Jeżeli zmiany implementacji klasy nie pociągają za sobą zmian nagłówków metod, to nie jest wymagana zmiana reszty kodu aplikacji.

Mechanizmy hermetyzacji danych ułatwiają panowanie nad integralnością danych przechowywanych przez obiekty a dziedziczenie tworzenie nowych klas o funkcjonalnościach zbliżonych do już zaimplementowanych.

Prześledzimy podstawowe zasady programowania obiektowego na przykładzie klasy reprezentującej człowieka. W aplikacji potrzebne jest przechowywanie imienia, wieku i wzrostu człowieka. Będzie również potrzebna metoda obliczająca wagę osobnika oraz generująca tekstową reprezentację obiektu. Zaczniemy od osobnika rodzaju męskiego, od stworzenia szkieletu klasy zawierającego potrzebne dane:

```
class Facet{
     protected String imie;
     protected double wiek, wzrost;
}
```

W klasie **`Facet`** zadeklarowano trzy pola: **`imie`** – łańcuch tekstowy oraz **`wiek`** i **`wzrost`**, jako liczby rzeczywiste. Metody, które zawsze powinna posiadać konstruowana klasa,

to konstruktor (lub konstruktory). Konstruktor jest metodą publiczną, której nazwa jest taka sama, jak nazwa klasy. Dla konstruktora nie określamy typu wartości funkcji. Wartością konstruktora jest adres utworzonego obiektu. Zadaniem konstruktora jest właściwe (zgodnie z życzeniem programisty) zainicjalizowanie danych przechowywanych przez obiekt (a tak naprawdę „włączenie" mechanizmu polimorfizmu, ale o tym będzie później). Oto klasa **`Facet`** uzupełniona o dwa konstruktory:

```
class Facet{
     public Facet(){
          imie = "NoName";
          wiek = 0;
          wzrost = 50;
     }
     public Facet(String imie, double wiek, double wzrost){
          this.imie = imie;
          this.wiek = wiek;
          this.wzrost = wzrost;
     }
     protected String imie;
     protected double wiek, wzrost;
}
```

Oba konstruktory mają taką samą nazwę. Cecha ta nazwana jest przeciążaniem metody. Podczas wykonywania programu wywoływana jest odpowiednia metoda ze zbioru metod przeciążonych na podstawie aktualnych parametrów. Każdy wariant przeciążonej metody musi w związku z tym różnić się parametrami, co do ich liczby lub typu. Dla naszej klasy moglibyśmy więc zdefiniować jeszcze następujące warianty konstruktorów:

```
public Facet(String imie, double wiek)
public Facet(double wiek, double wzrost)
```

Oba warianty różnią się od poprzednio zaprezentowanych liczbą parametrów (mają po 2, a wcześniejsze odpowiednio 0 i 3). Rozróżnienie pomiędzy tymi dwoma wariantami następuje na podstawie typu parametrów. Pierwszy ma parametry typu **`String`** oraz **`double`**, drugi **`double`** oraz **`double`**. Przy tak zdefiniowanych czterech konstruktorach nie jest możliwe zdefiniowanie 5 wariantu, w postaci:

```
public Facet(String imie, double wzrost)
```

ponieważ jest on nierozróżnialny z wariantem:

```
public Facet(String imie, double wiek)
```

- oba mają dwa parametry, jeden typu **`String`**, drugi **`double`**.

Zdefiniujmy w naszej klasie teraz kolejne metody tj. **`waga()`** i **`toString()`**.

```
class Facet{
     public Facet(){
          imie = “NoName”;
          wiek = 0;
          wzrost = 50;
     }
     public Facet(String imie, double wiek, double wzrost){
          this.imie = imie;
          this.wiek = wiek;
          this.wzrost = wzrost;
     }
     public toString(){
          String txt = “Nazywam się” + imie + “, mam ” + wiek + “ lat”;
          return txt;
     }
     public double waga(){
          return wzrost – 110 + 0.1 * wiek;
     }
     protected String imie;
     protected double wiek, wzrost;
}
```

Jest to kompletna już klasa zapewniająca ustaloną wcześniej funkcjonalność obiektu reprezentującego mężczyznę. Zajmijmy się teraz klasą reprezentującą kobietę. Będzie się ona różnić od klasy **`Facet`** tylko funkcją obliczającą wagę, wobec tego nie trzeba jej pisać całkowicie od nowa, tylko zdefiniujemy ją na bazie klasy **`Facet`**, korzystając z możliwości dziedziczenia.

```
class Baba extends Facet{
     public Baba(){
          imie = "NoName";
          wiek = 0;
          wzrost = 50;
     }
     public Baba(String imie, double wiek, double wzrost){
          this.imie = imie;
          this.wiek = wiek;
          this.wzrost = wzrost;
     }
     @Override
     public double waga(){
          return wzrost – 120 + 0.2 * wiek;
     }
}
```

Klasa **`Baba`** dziedziczy po klasie **`Facet`** wszystkie elementy składowe: pola i metody, za wyjątkiem konstruktorów. Dlatego należy konstruktory zdefiniować ponownie. Zgodnie za założeniem jest również nowy kod metody **`waga()`** obliczający wagę wg innego wzoru.

Nasz przykład nie jest doskonały. Zaraz wykażemy dlaczego. Spróbujmy rozbudować klasę **Baba** o metodę tworzącą dziecko. Problemem jest określenie typu wartości takiej metody. Dziecko może być płci męskiej i żeńskiej. Dla naszej implementacji należałoby zdefiniować dwie metody, każdą z innym typem wartości (**Facet**, **Baba**). Nie jest to dobre rozwiązanie. Błąd został popełniony podczas określania hierarchii klas. Zamiast tworzyć klasę **Baba** na bazie klasy **Facet** lepiej byłoby wyprowadzić te dwie klasy z klasy **Czlowiek**, w której zdefiniujemy wspólne metody oraz pola. Ponieważ nie jest dopuszczalne tworzenie obiektów klasy **Czlowiek**, lecz tylko obiekty określonej płci, stworzymy klasę **Czlowiek** jako klasę abstrakcyjną. Oto zmodyfikowany zestaw klas:

```
abstract class Czlowiek{
     public abstract double waga();
     public toString(){
          String txt = "Nazywam się" + imie + ", mam " + wiek + " lat";
          return txt;
     }
     protected String imie;
     protected double wiek, wzrost;
}

class Facet extends Czlowiek{
     public Facet(){
          imie = "NoName";
          wiek = 0;
          wzrost = 50;
     }
     public Facet(String imie, double wiek, double wzrost){
          this.imie = imie;
          this.wiek = wiek;
          this.wzrost = wzrost;
     }
     @Override
     public double waga(){
          return wzrost - 110 + 0.1 * wiek;
     }
     protected String imie;
     protected double wiek, wzrost;
}
class Baba extends Czlowiek{
     public Baba(){
          imie = "NoName";
          wiek = 0;
          wzrost = 50;
     }
     public Baba(String imie, double wiek, double wzrost){
          this.imie = imie;
          this.wiek = wiek;
          this.wzrost = wzrost;
     }
     public Czlowiek dziecko(Facet ojciec){
          String noweImie = imie+ojciec.imie;
          java.util.Random r = new java.util.Random();
          int plec = r.nextInt(2);
```

```
            Czlowiek potomek;
            if(plec=0) potomek = new Facet(noweImie,0,50);
               else potomek = new Baba(noweImie,0,50);
            return potomek;
        }
        @Override
        public double waga(){
            return wzrost - 120 + 0.2 * wiek;
        }
    }
```

W ten sposób możemy już użyć metody dziecko dla obu płci. W metodzie zastosowano losowe wybieranie płci dla dziecka. Czy dziecko jest typu **`Facet`** czy **`Baba`** sprawdzamy przynależność do kasy za pomocą konstrukcji **`if(obiekt instanceof klasa) {...}`**.

Mechanizm wywoływania odpowiedniej implementacji metody w zależności od typu obiektu jest nazywany polimorfizmem metod. Zapoznajmy się z tym mechanizmem na poniższym przykładzie:

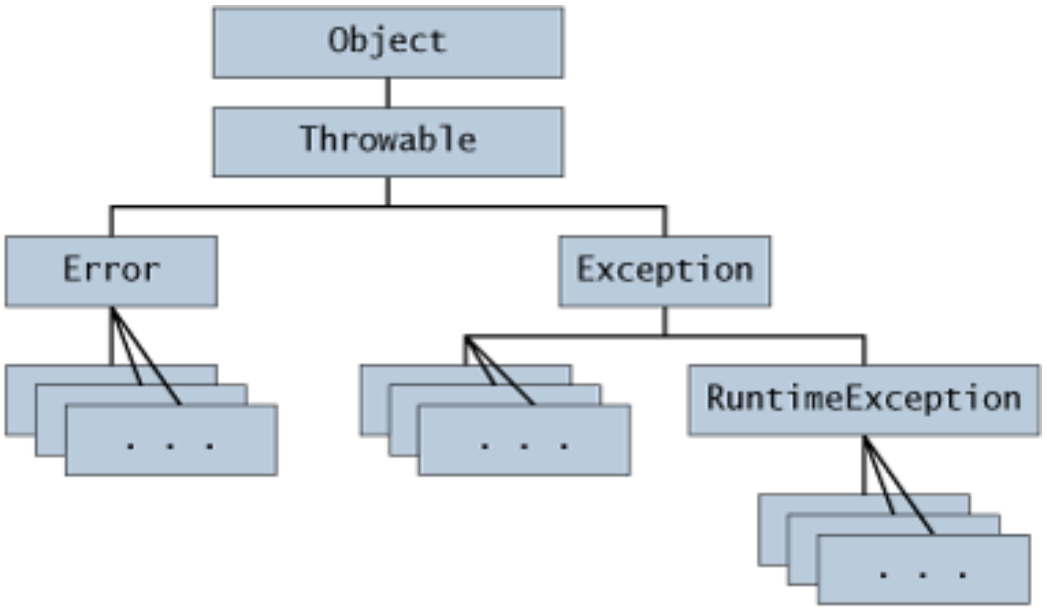
```
Czlowiek[] rodzina = new Czlowiek[5];
rodzina[0] = new Facet("Ojciec",40,180);
rodzina[1] = new Baba("Matka",35,160);
rodzina[2] = new Facet("PierwszySyn",10,140);
rodzina[3] = new Baba("Córka",8,130);
rodzina[4] = new Facet("DrugiSyn",6,120);
for(int i=0; i<rodzina.length;i++)
    System.out.println(rodzina[i].toString()+" "+rodzina[i].waga();
```

W pętli **`for`** funkcja **`waga()`** jest wywoływana poprzez identyfikator klasy **`Czlowiek`**. Ponieważ metoda ta jest różnie zaimplementowana w klasach potomnych **`Facet`** i **`Baba`**, maszyna Javy wykonuje odpowiednią implementację tej metody w zależności od typu obiektu znajdującego się na danej pozycji w tablicy.

W następnym rozdziale omówimy metody testowania i reagowania w programie na nieoczekiwane zdarzenia, czyli wyjątki.

Wyjątki

Podczas wykonywania każdego programu mogą zachodzić zdarzenia, które nie są zdarzeniami zachodzącymi podczas standardowego przetwarzanie programu. Takimi zdarzeniami może być sygnalizacja braku gotowości urządzeń lub próba wykonania instrukcji w danej sytuacji niedopuszczalnej. W języku Java przewidziano mechanizm obsługi tego typu zdarzeń nadając im przy tym nazwę wyjątków. Wyjątek w Javie jest obiektem, który jest powoływany do życia aby poinformować inne obiekty o zaistniałej sytuacji „wyjątkowej". Poniższy rysunek przedstawia hierarchię klas wyjątków:



W rozdziale tym zajmiemy się zarówno obsługą wyjątków zgłaszanych przez inne obiekty jak i mechanizmem zgłaszania wyjątków w projektowanych przez nas klasach.

Cytując za Brucem Eckelem wyjątków należy używać do:

- naprawiania problemów na odpowiednim poziomie (należy unikać przechwytywania wyjątków, jeśli nie wiadomo, co z nimi zrobić),
- naprawienia problemu i ponownego wywołania metody, która spowodowała wyjątek,
- wyjścia z sytuacji, która spowodowała wyjątek, i kontynuowania bez ponownego wywołania szwankującej metody,

- wygenerowania alternatywnego rozwiązania zamiast tego, które miała wyprodukować metoda,
- zrobienia, co tylko się da w aktualnym kontekście, i zgłoszenia ponownie tego samego wyjątku do kontekstu nadrzędnego,
- zrobienia, co tylko się da w aktualnym kontekście, i zgłoszenia ponownie innego wyjątku do kontekstu nadrzędnego.

1. Obsługa wyjątku zgłaszanego podczas wykonywania programu.

Pisząc kod aplikacji należy przewidywać sytuacje, w których może dochodzić do zgłoszenia wyjątku. Instrukcje, gdzie przewidujemy taką sytuację zamykamy w blok **`try…catch`**:

```
try {
     Kod programu mogący generować wyjątki
} catch (TypWyjątku1 e){
     Obsługa wyjątku 1
} catch (TypWyjątku2 e){
     Obsługa wyjątku 2
}
//...
finally {
     Blok instrukcji, który wykona się zawsze, niezależnie od
     wystąpienia lub nie wyjatków
}
```

Rozważmy następujący prosty przypadek – wykonanie operacji dzielenia dwóch liczb całkowitych. Jest ona dopuszczalna pod warunkiem, że dzielnik jest różny od zera. Poniższy kod spowoduje wygenerowanie wyjątku:

```
int a = 1234; int b = 0;
int wynik = a / b;
```

Program zostanie przerwany przez maszynę Javy, na konsoli zostanie wyświetlony komunikat o wystąpieniu wyjątku arytmetycznego: błąd dzielenia przez zero. Aby temu zapobiec powinniśmy sami obsłużyć w naszym programie tę sytuację. Obsługa będzie polegała na zastąpieniu komunikatu wyjątku własnym.

```
int a = 1234; int b = 0;
int wynik;
try {
     wynik = a / b;
} catch (ArithmeticException e) {
     System.out.println("Pamietaj cholero nie dziel przez zero");
}
```

Możemy w jednym bloku obsłużyć więcej niż jeden typ wyjątku. Spróbujmy zapisać to w ten sposób:

```
int a = 1234; int b = 0;
int wynik;
try {
     wynik = a / b;
} catch (Exception e) {
     System.out.println("Wystąpił nieznany błąd");
} catch (ArithmeticException e) {
     System.out.println("Pamietaj cholero nie dziel przez zero");
}
```

Powyższy kod jest błędny. Przy tak zapisanej kolejności obsługi wyjątków zawsze, gdy wystąpi jakikolwiek wyjątek pojawi się komunikat „Wystąpił nieznany błąd”. Musimy tutaj pamiętać o hierarchii klas. W instrukcji **`catch`** porównywany jest bieżący typ wyjątku z typem umieszczonym wewnątrz nawiasów. **`ArithmeticException`** jako potomek klasy **`Exception`** też jest typu **`Exception`** stąd wybieranie tego bloku obsługi wyjątku. Należy zastosować kolejność począwszy od klasy najbardziej potomnej. Oto poprawny kod:

```
int a = 1234; int b = 0;
int wynik;
try {
     wynik = a / b;
} catch (ArithmeticException e) {
     System.out.println("Pamietaj cholero nie dziel przez zero");
} catch (Exception e) {
     System.out.println("Wystąpił nieznany błąd");
}
```

2. Definiowanie i zgłaszanie własnych wyjątków

Ponieważ wyjątek jest obiektem należy zdefiniować klasę dziedziczącą po innej klasie będącej wyjątkiem, np.:

```
public class MojWyjatek extends Exception{
}
```

Taka definicja wystarcza, aby zgłaszać wyjątki z identyfikatorem **`MojWyjatek`**.

Implementując własną funkcję, która będzie zgłaszać wyjątek, musimy poinformować o tym już w nagłówku tej metody:

```
public void metodaZWyjatkiem() throws MojWyjatek {…};
```

Wszystkie wyjątki, które będzie zgłaszać metoda należy wymienić po słowie kluczowym **`throws`**. Może być ich więcej niż jeden. Wskazanie wyjątków w nagłówku metody wymusza później ich obsługę podczas wywoływania tej metody.

Przykład:

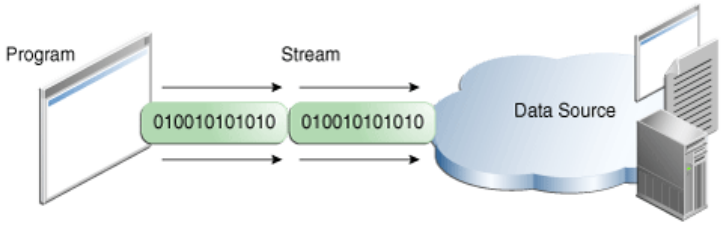
```
class BledneDane extends Exception{
}

class Ulamek{
   public Ulamek(int licznik, int mianownik) throws BledneDane{
      if(mianownik==0) throw new BledneDane();
      else{
         this.licznik=licznik;
         this.mianownik=mianownik;
      }
   }
   int licznik;
   int mianownik;
}

class Zadanie{
   public static void main(String[] args){
      Ulamek ulamek1,ulamek2;
      try {
         ulamek1 = new Ulamek(1,2);
         ulamek2 = new Ulamek(3,0);
      } catch(BledneDane bd){
         System.out.println("Mianownik ułamka musi byc różny od zera");
      }
   }
}
```

W powyższym przykładzie próba utworzenia obiektu **`ulamek2`** wygeneruje wyjątek o nazwie **`BledneDane`** oraz wyświetlenie informacji zawartej w klauzuli **`catch`**.

Strumienie

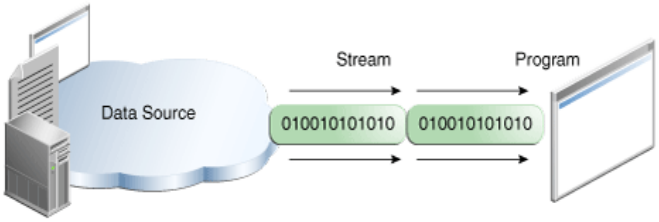
Strumień wejścia/wyjścia reprezentuje źródło/przeznaczenie do przesyłu danych na zewnątrz programu. Źródłem/przeznaczeniem może być:

- plik dyskowy,
- urządzenie,
- inny program,
- tablica w pamięci,
- gniazdo sieciowe,
- ...



Strumienie obsługują różne rodzaje danych:

- pojedyncze bajty,
- pierwotne typy danych,
- ciągi znaków,
- obiekty.



Niektóre strumienie jedynie przesyłają dane, inne wykonują na nich użyteczne czynności (np. różnego rodzaju konwersje). Strumienie reprezentują taki sam prosty model dla programów niezależnie od swego wewnętrznego działania – strumień jest sekwencją danych; program używa strumienia wejściowego do czytania danych ze źródła i strumienia wyjściowego do pisania danych w miejsce przeznaczenia (każda operacja przesyła jeden element).

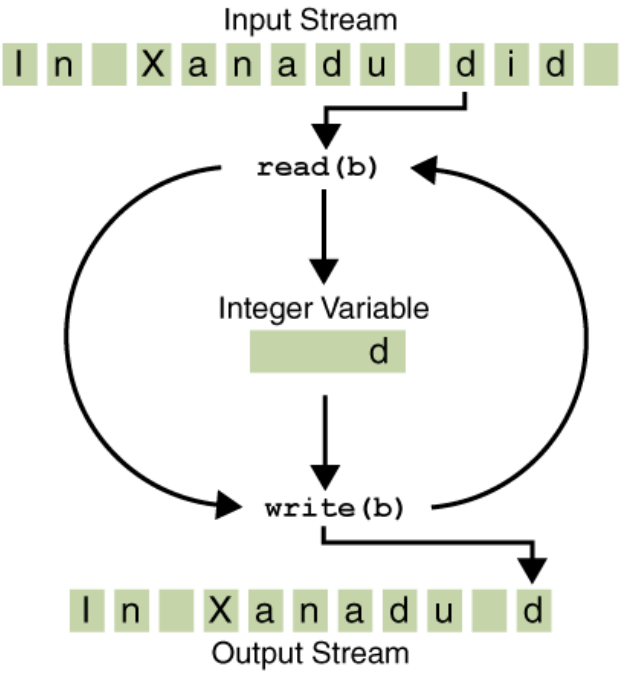
Strumienie bajtowe:

- Programy używają strumieni bajtowych do przesyłania porcji 8-bitowych (bajtów).
- Wszystkie klasy strumieni bajtowych są potomkami klas `InputStream` lub `OutputStream`.
- Na przykład do przetwarzania plików służą klasy `FileInputStream` oraz `FileOutputStream`.

- Istnieje wiele klas dla strumieni, ale wszystkich używa się podobnie; różnią się jedynie swoją budową wewnętrzną.

Przykład użycia strumieni bajtowych (kopiowanie pliku bajt po bajcie):

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes
{ public static void main(String[] args) throws IOException
  { FileInputStream  in  = null;
    FileOutputStream out = null;
    try
    { in  = new FileInputStream("xanadu.txt");
      out = new FileOutputStream("outagain.txt");
      int c; // przechowuje wartości na ostatnim bajcie

      while ((c = in.read()) != -1) // read() zwraca wartość
      out.write(c);                 // typu int dlatego c
                                      // jest typu int
    }                                 // (-1 to koniec pliku)
    finally
    { if (in  != null) in.close();
      if (out != null) out.close();
    }
  }
}
```



Zawsze należy zamykać strumienie (pliki) – jest to ważne, aby uniknąć utraty danych. Dobrą praktyką jest zamykanie strumieni w części **`finally`** z uwzględnieniem wartości różnej od **`null`** (nadanej jawnie przy tworzeniu strumienia). W zasadzie nie należy używać strumieni bajtowych, gdyż jest to programowanie „niskopoziomowe". Do obsługi różnych typów danych służą różne typy strumieni (np. do przesyłania znaków – strumienie znakowe), jednak wszystkie strumienie są zbudowane na bazie strumieni bajtowych.

Strumienie znakowe

Java zapisuje wartości znakowe używając UNICODE

- Strumienie znakowe automatycznie konwertują format wewnętrzny znaków na lokalny zbiór znaków używany w systemie (np. na nadzbiór kodu ASCII).
- Dla większości aplikacji praca ze strumieniami znakowymi niewiele różni się od pracy ze strumieniami bajtowymi – odpowiednie klasy dokonują automatycznych konwersji na lokalne zestawy znaków.
- Wszystkie klasy strumieni znakowych są potomkami klas **`Reader`** lub **`Writer`**.
- Do pracy z plikami służą klasy **`FileReader`** oraz **`FileWriter`**.

Przykład (kopiowanie pliku znak po znaku)

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters
{ public static void main(String[] args) throws IOException
  { FileReader inputStream  = null;
    FileWriter outputStream = null;

    try
    { inputStream  = new FileReader("xanadu.txt");
      outputStream = new FileWriter("characteroutput.txt");

      int c;    // przechowuje wartości na ostatnich 2 bajtach
      while ((c = inputStream.read()) != -1)  // jak w poprzednim
                                                 przykładzie
      outputStream.write(c);
    }
    finally { if (inputStream  != null) inputStream.close();
      if (outputStream != null) outputStream.close();
```

```
        }
      }
    }
```

Strumienie znakowe używają strumieni bajtowych do wykonania operacji fizycznych (niskopoziomowych), a same dokonują konwersji pomiędzy znakami a bajtami. **`FileReader`** używa strumienia **`FileInputStream`**; **`FileWriter`** używa strumienia **`FileOutputStream`**;

Przetwarzanie strumienia znakowego wierszami, gdzie wiersz (linia) – łańcuch znaków zakończony znacznikiem końca linii. Znacznikiem końca linii może być:

- **`"\r\n"`** (CRLF)
- **`"\r"`** (CR)
- **`"\n"`** (LF)

Dzięki rozpoznawaniu różnych znaczników końca linii otrzymuje się możliwość czytania plików utworzonych w różnych systemach operacyjnych (niezależność od platformy – przenośność).

Do przetwarzania pliku wierszami stosuje się metody:

- **`BufferedReader.readLine()`** – rozpoznaje znacznik końca linii
- **`PrintWriter.println()`** – dopisuje znacznik końca linii

Przetwarzanie pliku wierszami – przykład:

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines
{ public static void main(String[] args) throws IOException
  { BufferedReader inputStream = null;
    PrintWriter outputStream = null;

    try
    { inputStream  = new BufferedReader(
                     new FileReader("xanadu.txt"));
      outputStream = new PrintWriter(
                     new FileWriter("characteroutput.txt"));

      String l;
      while ((l = inputStream.readLine()) != null)
        outputStream.println(l);
```

```
    }
    finally
    { if (inputStream  != null) inputStream.close();
      if (outputStream != null) outputStream.close();
    }
  }
}
```

Buforowanie strumieni

Używając strumieni niebuforowanych każde żądanie czytania/zapisu jest związane z dostępem fizycznym do danych realizowanym przez system operacyjny, co jest mało efektywne. Aby tego uniknąć stosuje się strumienie buforowane.Buforowany strumień wejściowy czyta dane z bufora (specjalnego obszaru pamięci); dostęp fizyczny do strumienia odbywa się tylko wtedy, gdy bufor jest pusty. Buforowany strumień wyjściowy pisze dane do bufora w pamięci; dostęp fizyczny do strumienia odbywa się wtedy, gdy bufor jest pełny.Buforowanie uzyskuje się „opakowując" strumień niebuforowany (np. FileReader) strumieniem buforowanym (Bufferedreader) przekazując obiekt reprezentujący strumień niebuforowany do konstruktora strumienia buforowanego:

```
inputStream  = new BufferedReader(
                 new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(
                 new FileWriter("characteroutput.txt"));
```

Strumienie buforowane:

- **`BufferedInputStream`** i **`BufferedOutputStream`** – dla strumieni bajtowych
- **`BufferedReader`** i **`BufferedWriter`** – dla strumieni znakowych

W razie potrzeby bufor strumienia wyjściowego można „wymieść" (natychmiast przenieść jego zawartość do strumienia fizycznego) bez czekania na jego zapełnienie za pomocą metody **`flush()`**. Niektóre buforowane strumienie wyjściowe posiadają opcję automatycznego wymiatania (włączaną opcjonalnym argumentem) – wtedy niektóre zdarzenia powodują automatyczne wywołanie metody **`flush()`**. Dla klasy

PrintWriter autowymiatanie uruchamiane jest po każdym wywołaniu metody **println()** lub **format()**

Używając strumieni niebuforowanych każde żądanie czytania/zapisu jest związane z dostępem fizycznym do danych realizowanym przez system operacyjny, co jest mało efektywne. Aby tego uniknąć stosuje się strumienie buforowane. Buforowany strumień wejściowy czyta dane z bufora (specjalnego obszaru pamięci); dostęp fizyczny do strumienia odbywa się tylko wtedy, gdy bufor jest pusty. Buforowany strumień wyjściowy pisze dane do bufora w pamięci; dostęp fizyczny do strumienia odbywa się wtedy, gdy bufor jest pełny.

Buforowanie uzyskuje się „opakowując" strumień niebuforowany (np. **FileReader**) strumieniem buforowanym (**BufferedReader**) przekazując obiekt reprezentujący strumień niebuforowany do konstruktora strumienia buforowanego:

Strumienie danych

Strumienie danych wspomagają binarne wejście/wyjście w obsłudze pierwotnych typów danych: **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** i **double** oraz łańcuchów typu **String**. Strumienie te implementują interfejsy **DataInput** lub **DataOutput** w klasach:

- **DataInputStream**
- **DataOutputStream**

Do przesyłania danych do/ze strumienia służą metody:

- **writeBoolean() / readBoolean()**
- **writeByte() / readByte()**
- **writeChar() / readChar()**
- **writeInt() / readInt()**
- **writeDouble() / readDouble()**
- **writeUTF() / readUTF()**
- .....

UWAGA

Nie należy stosować metody **readLine()** z klasy **DataInputStream** (może błędnie konwertować bajty na znaki); zastępuje ją metoda **readLine()** klasy **BufferedReader**

Przykład (zapis i odczyt rekordów danych):

Każdy rekord danych składa się z trzech pól: **`cena, liczba, opis`**:

Pozycja w rekordzie	Typ pola	Opis pola	Matoda zapisująca	Matoda odczytująca	Przykładowa wartość
1	double	cena produktu	DataOutputStream .writeDouble()	DataInputStream. readDoubl()e	19.99
2	int	liczba sztuk	DataOutputStream .writeInt()	DataInputStream. readInt()	12
3	String	opis produktu	DataOutputStream .writeUTF()	DataInputStream. readUTF()	"Java T-Shirt"

```
// Definicje początkowe
static final String plik = "Dane";

static final double[] ceny   = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[]    sztuki = { 12, 8, 13, 29, 50 };
static final String[] opisy  = { "Java T-shirt",
                                 "Java Mug",
                                 "Duke Juggling Dolls",
                                 "Java Pin",
                                 "Java Key Chain" };
//...........
// Otwarcie pliku (strumienia) do zapisu
DataOutputStream out = new DataOutputStream(
                         new BufferedOutputStream(
                           new FileOutputStream( plik )));

// Zapis danych do pliku
for (int i = 0; i < ceny.length; i ++)
{ out.writeDouble(ceny[i]);
  out.writeInt(sztuki[i]);
  out.writeUTF(opisy[i]);
}
out.close();
//...
// Otwarcie pliku do odczytu
DataInputStream in = new DataInputStream(
                        new BufferedInputStream(
                          new FileInputStream(dataFile)));

// Deklaracje elementów rekordu
double cena;
int    sztuk;
String opis;
double suma = 0.0;
// Odczyt danych
try
{ while (true)
  { cena  = in.readDouble(); // uwaga na kolejność danych
    sztuk = in.readInt();
    opis  = in.readUTF();
```

```
    System.out.println("Zamówiłeś " + sztuk + " sztuk produktu " + opis + "
po " + cena);
    suma += sztuk * cena;
  }
}
catch (EOFException e) // koniec pliku
{ in.close();
  System.out.println("Zapłaciłeś " + suma);
}
/*************************************************************
Zamowiles 12 sztuk produktu Java T-shirt po 19.99
Zamowiles 8 sztuk produktu Java Mug po 9.99
Zamowiles 13 sztuk produktu Duke Juggling Dolls po 15.99
Zamowiles 29 sztuk produktu Java Pin po 3.99
Zamowiles 50 sztuk produktu Java Key Chain po 4.99
Zaplaciles 892.8800000000001
*************************************************************/
```

UWAGA

Nie należy stosować typów zmiennoprzecinkowych do reprezentacji wartości walutowych – należy zastosować typ **`BigDecimal`**. Niestety jest to typ obiektowy i jego wartości nie mogą być zapisywane za pomocą strumieni danych (służy do tego strumienie obiektów).

Strumienie obiektów

Służą do obsługi obiektów. Potrzebna jest serializacja obiektów – proces przekształcania obiektów do postaci sekwencyjnej, czyli na strumień bajtów, z zachowaniem aktualnego stanu obiektu; służy do zapisu obiektu (np. do strumienia), a później do jego odtworzenia.

Klasy do obsługi strumieni obiektów:

- **`ObjectInputStream`**
- **`ObjectOutputStream`**

Implementują podinterfejsy **`ObjectInput`** i **`ObjectOutput`** interfejsów **`DataInput`** i **`DataOutput`**. Oprócz metod obsługujących obiekty:

- **`readObject()`**
- **`writeObject()`**

Zawierają także metody operujące na pierwotnych typach danych – w strumieniu można mieszać wartości typów pierwotnych i obiektowych.

Jeżeli metoda **`readObject()`** nie zwróci obiektu odpowiedniego typu, powstaje wyjątek **`ClassNotFoundException`**.

Przykład:

```
import java.io.*;
import java.math.BigDecimal;

public class Main
{ static final String plik = "Dane";

  static final BigDecimal[] ceny =
                               { new BigDecimal("19.99"),
                                 new BigDecimal("9.99"),
                                 new BigDecimal("15.99"),
                                 new BigDecimal("3.99"),
                                 new BigDecimal("4.99")
                               };
  static final int[]    sztuki = { 12, 8, 13, 29, 50 };
  static final String[] opisy  = { "Java T-shirt",
                                   "Java Mug",
                                   "Duke Juggling Dolls",
                                   "Java Pin",
                                   "Java Key Chain" };

  public static void main( String[] args )
        throws IOException, ClassNotFoundException
  { ObjectOutputStream out = null;
    try
    { out = new ObjectOutputStream(
              new BufferedOutputStream(
                new FileOutputStream( plik )));

      for ( int i = 0; i < ceny.length; i ++ )
      { out.writeObject( ceny[i] );
        out.writeInt( sztuki[i] );
        out.writeUTF( opisy[i] );
      }
    }
    finally { out.close(); }
    ObjectInputStream in = null;
    try
    { in = new ObjectInputStream(
             new BufferedInputStream(
               new FileInputStream( plik )));
      BigDecimal cena;
      int        sztuk;
      String     opis;
      BigDecimal suma = new BigDecimal( 0 );

      try
      { while (true)
        { cena = (BigDecimal) in.readObject();
          sztuk = in.readInt();
          opis = in.readUTF();
          System.out.println("Zamówiłeś " + sztuk + " sztuk produktu " +
                             opis + " po " + cena );
```

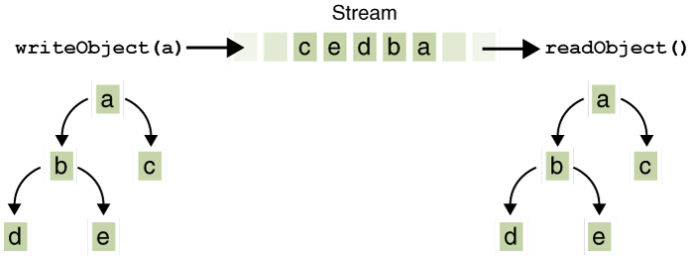
```
            suma = suma.add( cena.multiply( new BigDecimal( sztuk )));
          }
        }
        catch ( EOFException e ) {}
        System.out.println("Zapłaciłeś " + suma );
      }
      finally { in.close(); }
    }
}
```

Pisanie i czytanie w strumieniach obiektów – uwagi

Metody **`writeObject()`** i **`readObject()`** oprócz serializacji obiektów wykrywają także pola będące referencjami do obiektów:

- metoda **`writeObject()`** zapisuje do strumienia nie tylko obiekt określony parametrem, ale wszystkie obiekty z nim związane „wychodzącymi" z niego referencjami
- metoda **`readObject()`** odtwarza strukturę obiektów zapisanych w strumieniu przez metodę **`writeObject()`**

Należy więc pamiętać, że jedno wywołanie metody **`writeObject()`** może spowodować zapisanie do strumienia dużej liczby obiektów.



Jeżeli do strumienia zostanie wielokrotnie zapisany ten sam obiekt, to po odczytaniu powstanie tylko jeden jego egzemplarz:

```
Object ob = new Object();
out.writeObject( ob );
out.writeObject( ob );
.........
Object ob1 = in.readObject(); // obie zmienne zawierają referencję do tego
samego obiektu
Object ob2 = in.readObject();
```

Jeżeli ten sam obiekt zostanie zapisany do różnych strumieni, to po odczytaniu zostanie utworzonych wiele obiektów o tej samej wartości.

Pliki o dostępie bezpośrednim (klasa RandomAccessFile)

Obiekty klasy **`RandomAccessFile`** umożliwiają czytanie i zapis do plików o dostępie bezpośrednim (swobodnym). Plik o dostępie bezpośrednim zachowuje się jak tablica bajtów zapisana w systemie plików. Plik posiada rodzaj „kursora" (indeksu tablicy) nazywany wskaźnikiem pliku:

- operacje odczytu danych z pliku czytają bajty od pozycji wyznaczonej wskaźnikiem pliku i przesuwają go w trakcie czytania.

Jeżeli plik jest otwarty zarówno do odczytu, jak i do zapisu, można do niego wpisać dane (nadpisać istniejącą zawartość) – również począwszy od wskaźnika pliku, wskaźnik pliku jest automatycznie przesuwany za wpisaną zawartość. Jeżeli w trakcie pisania do pliku zostaje przekroczony jego koniec, plik jest automatycznie powiększany. Próba czytania poza końcem pliku skutkuje wygenerowaniem wyjątku **`EOFException`**. Wskaźnik pliku może być ustawiany metodą **`seek()`** oraz odczytany metodą **`getFilePointer()`.**

Przykład:

```
public class RandAccFile {

    public static void PiszPlik() throws IOException
    {
     RandomAccessFile plik = null;
     try{
         plik = new RandomAccessFile("Plik.bin", "rw");
         while( true )
           System.out.println( plik.readInt() );
        }
      catch(IOException e){ plik.close(); }
      System.out.println();
    }


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

```
    {
     try{
         File plikP = new File("Plik.bin");
         if(plikP.exists()) plikP.delete();
         RandomAccessFile plik = new RandomAccessFile("Plik.bin", "rw");

         for( int nr=0; nr<8; nr++ )
           plik.writeInt( nr );
         plik.close();

         System.out.println("Plik przed zmianami:");
         PiszPlik();

         // nadpisanie danych i dopisanie nowych
         plik = new RandomAccessFile("Plik.bin", "rw");
         plik.seek( 5*4 );
         for( int nr=55; nr<100; nr+=11)
           plik.writeInt( nr );
         plik.close();

         System.out.println("Plik po zmianach:");
         PiszPlik();

         System.out.println("Zawartość pliku w losowej kolejności:");
         plik = new RandomAccessFile("Plik.bin", "rw");
         for( int nr=1; nr<=10; nr++ )
         { int nrLicz = (int)( Math.random()*10 );
           plik.seek( nrLicz * 4 );
           System.out.println("Liczba nr " +nrLicz +": "+plik.readInt ());
         }
         plik.close();
        }
     catch ( IOException e ) { System.out.println("Błąd w pliku!!!"); }
    }
 }
```

Przykład 2:

```
public class RandAccFile1 {

  // Definicje początkowe
  static final String   plik   = "Dane";
  static final double[] ceny   = { 19.99, 9.99, 15.99, 3.99, 4.99 };
  static final int[]    sztuki = { 12, 8, 13, 29, 50 };
  static final String[] opisy  = { "Java T-shirt",
                                   "Java Mug",
                                   "Duke Juggling Dolls",
                                   "Java Pin",
                                   "Java Key Chain" };

 public static void main(String[] args) throws IOException{
   DataOutputStream out;
   DataInputStream  in = null;
   double cena;
   int sztuk;
   String opis;
   double suma = 0.0;
```

```
try{
      // Otwarcie pliku (strumienia) do zapisu
      out = new DataOutputStream(
              new BufferedOutputStream(
                new FileOutputStream( plik )));

      // Zapis danych do pliku
      for( int i = 0; i < ceny.length; i ++ )
      { out.writeDouble( ceny[i] );
        out.writeInt( sztuki[i] );
        out.writeUTF( opisy[i] );
      }
      out.close();

      // Otwarcie pliku do odczytu

      in = new DataInputStream(
             new BufferedInputStream(
               new FileInputStream( plik )));
      // Odczyt danych

      while( true )
      { cena  = in.readDouble(); // uwaga na kolejność danych
        sztuk = in.readInt();
        opis  = in.readUTF();
        System.out.println("Zamówiłeś " + sztuk + " sztuk produktu " +
                           opis + " po " + cena );
        suma += sztuk * cena;
      }
   }
catch( IOException e )
{ if( in != null ) in.close();
  System.out.println("Zapłaciłeś " + suma );
}

// Zmiana cen o 10% i nazw na duże litery

RandomAccessFile inOut = null;
long poz;
try{
     inOut = new RandomAccessFile( plik, "rw" );
     while( true )
     { cena = inOut.readDouble(); // uwaga na kolejność i rozmiar danych
       inOut.seek( inOut.getFilePointer() - 8 );
                           // cena (double) na 8 bajtach
       inOut.writeDouble( cena * 0.9 );   // obniżka o 10%
       sztuk = inOut.readInt();
       // poz = inOut.getFilePointer();   // można zapamiętać pozycję...
       opis  = inOut.readUTF().toUpperCase();
       inOut.seek( inOut.getFilePointer() - (opis.length() + 2 ));
                       // plus 2 bajty (długość)
       // inOut.seek(poz);   // ...i ją odtworzyć przed zapisem łańcucha
       inOut.writeUTF(opis);
      }
}
catch( EOFException e)
{ if ( inOut != null) in.close();
  System.out.println("Plik został zmodyfikowany");
}
```

```
   // Przepisanie pliku po zmianach

   try{
         in = new DataInputStream(
                new BufferedInputStream(
                  new FileInputStream( plik )));
         suma = 0;
         // Odczyt danych
         while (true)
         { cena  = in.readDouble(); // uwaga na kolejność danych
           sztuk = in.readInt();
           opis  = in.readUTF();
           System.out.println("Zamówiłeś " + sztuk + " sztuk produktu " +
                              opis + " po " + cena);
           System.out.format("Zamówiłeś %d sztuk produktu %s po %.2f\n\n",
                             sztuk, opis, cena);
           suma += sztuk * cena;
         }
      }
   catch( IOException e )
   { if (in != null) in.close();
     System.out.println("Zapłaciłeś " + suma);
     System.out.format("Zapłaciłeś %.2f\n", suma);
   }
  }
}
```

```
Zamówiłeś 12 sztuk produktu JAVA T-SHIRT po 17.991
Zamówiłeś 12 sztuk produktu JAVA T-SHIRT po 17,99

Zamówiłeś 8 sztuk produktu JAVA MUG po 8.991
Zamówiłeś 8 sztuk produktu JAVA MUG po 8,99

Zamówiłeś 13 sztuk produktu DUKE JUGGLING DOLLS po 14.391
Zamówiłeś 13 sztuk produktu DUKE JUGGLING DOLLS po 14,39

Zamówiłeś 29 sztuk produktu JAVA PIN po 3.591
Zamówiłeś 29 sztuk produktu JAVA PIN po 3,59

Zamówiłeś 50 sztuk produktu JAVA KEY CHAIN po 4.4910000000000005
Zamówiłeś 50 sztuk produktu JAVA KEY CHAIN po 4,49

Zapłaciłeś 803.5920000000001
Zapłaciłeś 803,59
```

Kolekcje

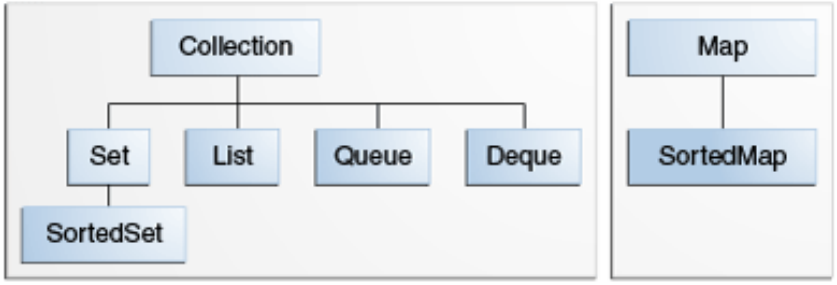
Kolekcja – zwana kontenerem – obiekt grupujący wiele elementów w pojedynczą jednostkę. Kolekcje służą do gromadzenia, udostępniania i manipulowania zagregowanymi danymi. Reprezentują elementy tworzące naturalne grupy – grupa kart (np. ręka w brydżu), zestaw listów elektronicznych (folder), książka telefoniczna (nazwiska i numery telefonów). Środowisko kolekcji (Collections Framework) to uniwersalna architektura do reprezentowania i manipulowania kolekcjami.

Środowiska kolekcji zawierają:

- Interfejsy – abstrakcyjne typy danych reprezentujące kolekcje. Pozwalają na operowanie kolekcjami nie zwracając uwagi na szczegóły związane z ich reprezentacją.
- Implementacje – konkretne implementacje interfejsów kolekcji. Są to gotowe do użycia struktury danych.
- Algorytmy – metody wykonujące użyteczne obliczenia – wyszukiwanie, sortowanie – na obiektach implementujących interfejsy związane z kolekcjami. Metody te są polimorficzne – ta sama metoda może być użyta dla wielu różnych implementacji odpowiedniego interfejsu kolekcji. Są zbiorem uniwersalnych funkcji.

Podstawowe interfejsy kolekcji hermetyzują różne typy kolekcji. Pozwalają operować na kolekcjach bez zwracania uwagi na szczegóły implementacje.

Podstawowe interfejsy związane z kolekcjami tworzą hierarchię:



Można zauważyć, że np. `Set` jest specjalnym rodzajem kolekcji `Collection`, `SortedSet` jest przypadkiem kolekcji `Set` itd. Na hierarchię składają się w rzeczywistości dwa rozłączne drzewa – `Map` nie jest prawdziwą kolekcją (`Collection`).

Java nie ma osobnych interfejsów dla każdego wariantu każdego typu kolekcji. Zamiast tego operacje są oznaczone, jako opcjonalne – dana implementacja nie wspiera wszystkich operacji, a w razie wywołania niezaimplementowanej operacji dla danej kolekcji jest generowany wyjątek: `UnsupportedOperationException`

Podstawowe interfejsy kolekcji są typami uogólnionymi (generycznymi):

```
public interface Collection<E>...
```

Kiedy deklaruje się instancję kolekcji (`Collection`) należy określić typ obiektu umieszczonego w kolekcji. Pozwala to kompilatorowi sprawdzić, czy obiekt umieszczanym w kolekcji jest odpowiedniego typu, co zmniejsza prawdopodobieństwo wystąpienia błędu w czasie wykonania programu.

Typ wyliczeniowy

Typ wyliczeniowy jest specjalnym typem zawierającym wartości ze zbioru predefiniowanych stałych. Ponieważ wartości tego typu są stałymi, zaleca się, żeby zapisywać je dużymi literami. Definicja takiego typu wykorzystuje słowo kluczowe **enum**:

```
public enum DniTygodnia {
    PONIEDZIAŁEK, WTOREK, ŚRODA, CZWARTEK, PIĄTEK, SOBOTA, NIEDZIELA
}
```

Typ wyliczeniowy służy do reprezentowania ustalonego zbioru stałych, które są znane już na etapie kompilacji, np. pozycje w menu, przełączniki, nazwy dni tygodnia, nazwy miesięcy itp.

```
public class TWyliczeniowe {

   DniTygodnia dzień;

     TWyliczeniowe(DniTygodnia dzień_) {
        dzień = dzień_;
    }


    public void JakToJest() {
        switch (dzień) {
            case PONIEDZIAŁEK:
                System.out.println("Nie lubię poniedziałku.");
                break;

            case PIĄTEK:
                System.out.println("W piątek już myślę o odpoczynku.");
                break;

            case SOBOTA: case NIEDZIELA:
                System.out.println("Najlepsze są popiątki.");
                break;

            default:
                System.out.println("Środek tygodnia można jakoś
                                                przeżyć.");
                break;
    }
}


    public static void main(String[] args) {
        TWyliczeniowe pierwszy =
                        new TWyliczeniowe(DniTygodnia.PONIEDZIAŁEK);
        pierwszy.PowiedzJakToJest();
        TWyliczeniowe drugi = new TWyliczeniowe(DniTygodnia.WTOREK);
```

```
        drugi.PowiedzJakToJest();
        TWyliczeniowe trzeci = new TWyliczeniowe(DniTygodnia.ŚRODA);
        trzeci.PowiedzJakToJest();
        TWyliczeniowe piąty = new TWyliczeniowe(DniTygodnia.PIĄTEK);
        piąty.PowiedzJakToJest();
        TWyliczeniowe szósty = new TWyliczeniowe(DniTygodnia.SOBOTA);
        szósty.PowiedzJakToJest();
        TWyliczeniowe siódmy = new TWyliczeniowe(DniTygodnia.NIEDZIELA);
        siódmy.PowiedzJakToJest();
    }
}
```

W Javie typy wyliczeniowe są bardziej zaawansowane niż w innych językach programowania. W rzeczywistości sekcja **`enum`** definiuje klasę. Kompilator, przy tworzeniu typu wyliczeniowego, tworzy (dodaje) kilka specjalnych metod, np.: metodę statyczną zwracającą tablicę zawierającą wszystkie wartości danego typu wyliczeniowego w kolejności ich deklarowania (która może być wykorzystana przez instrukcję for w wersji rozszerzonej):

```
System.out.printf("Kolejne dni tygodnia to:\n");
for (DniTygodnia dzień: DniTygodnia.values()) {
    System.out.printf(dzień + "\n");
}
```

UWAGA

Wszystkie typy wyliczeniowe, jako klasy, dziedziczą po klasie **`Java.lang.Enum`**, więc nie mogą dziedziczyć po innych klasach, bo w Javie nie ma dziedziczenia wielobazowego.

Wartości typu wyliczeniowego mogą być uzupełnione o dodatkowe stałe (właściwości). Właściwości te są przekazywane do konstruktora jako parametry w momencie tworzenia każdej stałej. Stałe (wartości typu wyliczeniowego) muszą być zdefiniowane na początku definicji **`enum`**. Po definicji stałych mogą wystąpić pola i metody (oddzielone od stałych wyliczeniowych średnikiem. Konstruktor musi być prywatny lub mieć dostępność pakietową (package-private). Konstruktor jest wywoływany automatycznie w momencie tworzenia stałych wyliczeniowych (nie może być wywoływany jawnie).

```
public enum Planety {
    MERKURY (3.303e+23, 2.4397e6),
    WENUS   (4.869e+24, 6.0518e6),
    ZIEMIA  (5.976e+24, 6.3781e6),
    MARS    (6.421e+23, 3.3972e6),
    JOWISZ  (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URAN    (8.686e+25, 2.5559e7),
    NEPTUN  (1.024e+26, 2.4746e7);

    private final double masa;    // w kilogramach
    private final double promień; // w metrach
    Planety(double masa, double promień) {
        this.masa = masa;
        this.promień = promień;
    }
    private double masa() { return masa; }
    private double promień() { return promień; }

    public static final double G = 6.67300E-11;
                // stała grawitacyjna  (m3 kg-1 s-2)

    double grawitacjaNaPlanecie() {
        return G * masa / (promień * promień);
    }
    double wagaNaPlanecie(double masa1) {
        return masa1 * grawitacjaNaPlanecie();
    }
    public static void main(String[] args) {
        double wagaNaZiemi = 75;
        double masa = wagaNaZiemi/ZIEMIA.grawitacjaNaPlanecie();
        for (Planety p : Planety.values())
           System.out.printf("Twoja waga na planecie %s wynosi
%f%n", p, p.wagaNaPlanecie(masa));
    }
}
```

Typy uogólnione (generyczne)

Java 5.0 wprowadziła typy uogólnione (generyczne). Przypominają (ale też są ważne różnice) szablony (templates) z C++. Typy uogólnione opisują abstrakcyjną stronę typów. Przykładem są kontenery (kolekcje) z hierarchii klas **`Collections`**. Typowe użycie kolekcji (bez typów uogólnionych):

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

Rzutowanie w linii 3 jest nieco „męczące”. Zwykle programista wie jakiego typu dane są umieszczane w konkretnej liście. Jednak rzutowanie jest niezbędne. Kompilator może jedynie zagwarantować, że iterator zwróci obiekt typu **`Object`**. Aby być pewnym, że przypisanie do zmiennej typu **`Integer`** jest bezpieczne, trzeba zastosować rzutowanie. Rzutowanie nie tylko wprowadza pewien bałagan, ale także może spowodować błąd wykonania w przypadku błędu programisty. Jeśli programista jest pewien, że lista będzie zawierała konkretny typ danych, wtedy może zastosować typ uogólniony (generyczny):

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

Należy zauważyć, że deklaracja zmiennej **`myIntList`** (linia 1) określa konkretny typ listy (**`LinkedList<Integer>`**), a nie listę ogólną (**`List<Integer>`**). Wtedy **`List`** jest interfejsem uogólnionym, który przyjmuje parametr związany z typem (w tym wypadku **`Integer`**). Taki sam parametr jest określony przy tworzeniu obiektu listy. W takim przypadku rzutowanie z linii 3 zostaje usunięte.

Motywacje:

- Zastosowanie typów uogólnionych nie polega jedynie na zamianie rzutowania na parametr.

- Zmiana jest dużo głębsza. Przy użyciu takiego typu kompilator sprawdza poprawność programu w czasie kompilacji.
- Dla dużych programów to rozwiązanie poprawia czytelność i pewność programu.

Definiowanie typów uogólnionych

Wycinek definicji interfejsów **`List`** oraz **`Iterator`** z pakietu **`Java.util`**:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

W nawiasach trójkątnych znajduje się deklaracja parametrów formalnych dla interfejsów. W odwołaniu do typu uogólnionego (nazywanego także typem sparametryzowanym) wszystkie wystąpienia parametru formalnego (typu) zostają zastąpione argumentem (parametrem aktualnym), np.:

```
List<Integer> myIntList = new LinkedList<Integer>();
```

Powoduje to przekształcenie:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

```
public interface List<Integer> {
    void add(Integer x);
    Iterator<Integer> iterator();
}

public interface Iterator<Integer>
{
    Integer next();
    boolean hasNext();
}
```

Czyli można sobie wyobrazić, że **List<Integer>** stanowi wersję **List**, w której parametr **E** został zastąpiony typem **Integer**. W rzeczywistości nie ma wielu kopii kodu – ani w kodzie źródłowym, ani wynikowym (w odróżnieniu od szablonów w C++). Deklaracja typu uogólnionego jest kompilowana jedynie raz i zapisywana w pliku **.class**, tak jak zwykła klasa.

```
public interface IntegerList {
    void add(Integer x);
    Iterator<Integer> iterator();
}
```

Parametry określające typy są traktowane analogicznie do zwykłych parametrów używanych w metodach – odpowiadają parametrom formalnym. W momencie wykorzystania deklaracji typu uogólnionego, parametry formalne są zastępowane parametrami aktualnymi określającymi aktualny typ.

Dziedziczenie

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

Linia pierwsza jest poprawna, ale linia druga wprowadza zamieszanie, bo w sytuacji:

```
lo.add(new Object());
String s = ls.get(0); // Attempts to assign an Object to
                      a String!
```

ponieważ **ls** i **lo** oznaczają tę samą listę, **ls** nie może przechowywać łańcuchów **String**. W związku z tym linia druga spowoduje błąd kompilacji.

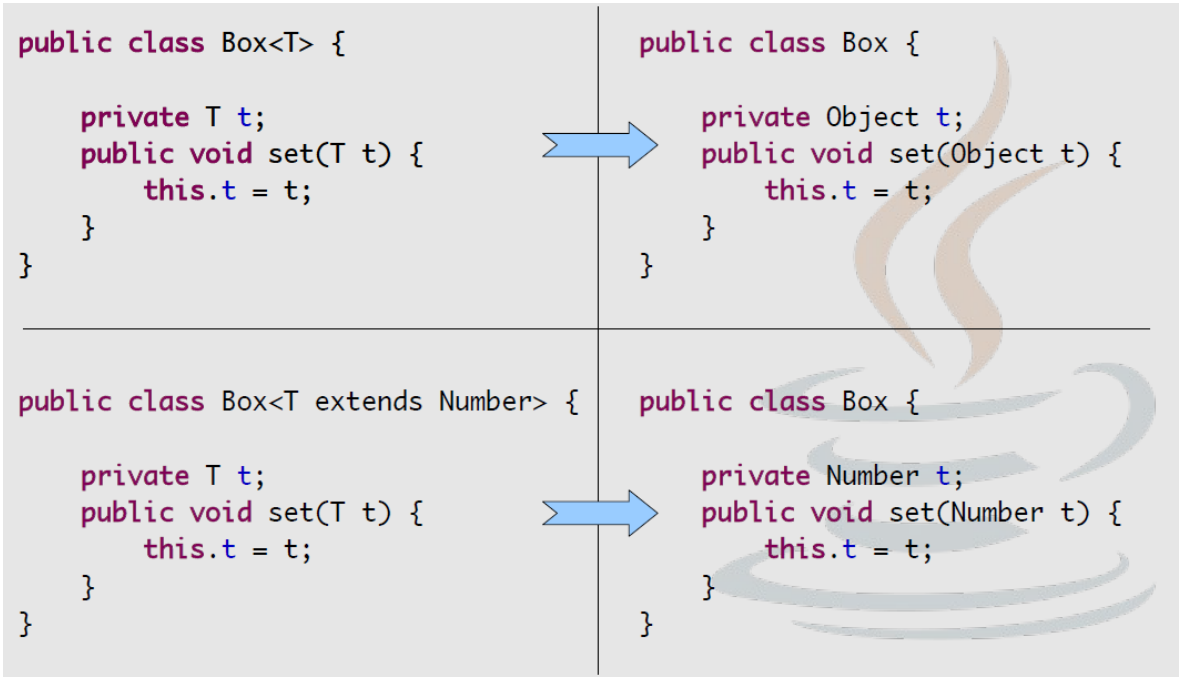
Ogólnie, jeżeli **Podtyp** jest podtypem (podklasą lub podinterfejsem) typu **Typ**, a **Ogólny** jest deklaracją typu uogólnionego, to **Ogólny<Podtyp>** nie jest podtypem typu **Ogólny<Typ>**. Na przykład, jeśli wydział komunikacji przekazuje listę kierowców do biura meldunkowego, wtedy **List<Driver>** jest tym samym co **List<Person>** (przyjmując, że **Driver** jest podtypem **Person**). Rzeczywiście, to co zostaje przesłane,

jest kopią rejestru kierowców. W przeciwnym razie biuro meldunkowe mogłoby dodać do listy nowe osoby, które nie są kierowcami, niszcząc dane w wydziale komunikacji.

Podsumowanie

Typy uogólnione (generyczne) zostały wprowadzone aby lepiej kontrolować typy na etapie kompilacji i aby umożliwić programowanie generyczne. W generowanym kodzie bajtowym otrzymuje się zwykłe klasy i interfersy. Aby to zrealizować kompilator:

- zastępuje typy przez ich ograniczenia a nieograniczone typy przez `Object`,
- dodaje niezbędne operacje rzutowania,
- tworzy metody pomostowe implementujące polimorfizm w rozszerzonych typach generycznych.



Ograniczenia

- nie można używać typów pierwotnych, np.: `Box<int>`,
- nie można używać operatora new, np.: `E e = new E()`,
- nie można deklarować typów statycznych, np.: `private static T v`,
- typów parametryzowanych (`List<Integer>`) nie można rzutować ani używać jako argument w operatorze `instanceof`,
- nie można używać tablic typów parametryzowanych, np.:

  `List<Integer>[] arrayOfLists = new List<Integer>[2]`

- typ generyczny nie może rozszerzać (bezpośrednio lub pośrednio) klasy `Throwable`,
- nie można przeciążać metod, których argumenty sprowadzają się do tego samego typu, np.:

```
public class Example {
   public void print(Set<String>  strSet) { }
   public void print(Set<Integer> intSet) { }
}
```

Zadania

1. Wyrażenia, instrukcje, metody
2. Klasy i obiekty
3. Obsługa wyjątków

4 Strumienie

5 Kontenery

6 Wielowątkowość

7 Programowanie sieciowe

Zadanie 1.1

Okna dialogowe są wygodnym sposobem interakcji programu z użytkownikiem. Kilka specjalizowanych rodzajów dialogów pozwala na wyświetlanie komunikatów, wprowadzanie danych czy uzyskiwanie potwierdzeń wykonywanych czynności od użytkownika. Do wyświetlania dialogów służą statyczne metody `showXXXDialog` z klasy `JOptionPane` pakietu `javax.swing`. Jest ich dość dużo, ze względu na bogate możliwości konfiguracji wyglądu i zachowań dialogów.

Napisz program, który:

1. za pomocą okna dialogowego pobierze od użytkownika łańcuch znakowy,
2. zamieni w nim małe litery na wielkie,
3. wyświetli wynik w dialogowym oknie informacyjnym.

Wskazówki

- Do wyświetlania okna dialogowego pozwalającego na wprowadzenie danych służy przeciążona, statyczna metoda `showInputDialog()` z klasy `JOptionPane`.
- Do zamiany małych liter na wielkie służy metoda `toUpperCase()` z klasy `String`. Zwraca ona nowy obiekt (nie modyfikuje źródła)!
- Do wyświetlenia okna informacyjnego służy statyczna, przeciążona metoda klasy `JOptionPane` o nazwie `showMessageDialog()`.

- Spośród wielu wariantów metod przeciążonych należy wybrać te o najmniejszej liczbie parametrów. Wykorzystują one domyślne wartości dla konfigurowalnych cech dialogów (np. ikona, tytuł).
- Programy korzystające z okien biblioteki `Swing` (w szczególności korzystające z dialogów wyświetlanych przez metody klasy `JOptionPane`) należy kończyć wywołując metodę `System.exit(int)`.

Zadanie 1.2.

Korzystając z klasy `BigInteger` napisać program, który wyznacza silnię z podanej liczby całkowitej n nawet dla dużych **n** (rzędu kilkuset).

Zadanie 1.3.

Napisać program sumujący liczby nieparzyste z przedziału od 1 do n, gdzie n - podaje użytkownik na starcie programu. Program powinien zakończyć sumowanie na liczbie n, gdy liczba n jest nieparzysta lub na liczbie n – 1, gdy liczba n jest parzysta.

Zadanie 1.4.

Operatory bitowe pozwalają traktować zmienne typów całkowitoliczbowych jak zestawy bitów i wykonywać na nich operacje. Oprócz bitowych odpowiedników operatorów logicznych (alternatywa, koniunkcja, negacja) dostępne są również operatory przesunięcia. Ich działanie polega na przesunięciu całego zestawu bitów o zadaną pozycję w lewo lub w prawo (część bitów zostanie utrącona, a wolne miejsca wypełnione 0 lub 1). Operatory bitowe stosuje się rzadko, głównie do kodowania w zwarty sposób binarnych informacji w postaci tzw. flag, które można badać za pomocą tzw. masek (są to zmienne typów całkowitoliczbowych z określonymi bitami ustalonymi na 1, a pozostałymi na 0). Chcemy mieć metody, które zmieniają liczbę na napis ją reprezentujący w zadanym systemie liczenia (binarny, ósemkowy, szesnastkowy). Aby uprościć zadanie, ograniczymy się do liczb nieujemnych. Zatem należy zaimplementować trzy metody pobierające jako argument liczbę całkowitą typu `int` i zwracającą łańcuch znakowy (obiekt klasy `String`) będący:

1. binarną reprezentacją argumentu,
2. ósemkową reprezentacją argumentu,
3. szesnastkową reprezentacją argumentu.

Wskazówki

• Można skorzystać z operatorów bitowych przesunięcia w prawo i koniunkcji.

• Można skorzystać z operatorów dzielenia oraz reszty z dzielenia (/, %).

Zadanie 2.1

Zaprojektuj klasę `Rational`, reprezentującą liczby wymierne jako pary liczb całkowitych (licznik i mianownik), wraz z podstawowymi działaniami arytmetycznymi i porównaniem. W klasie powinny znaleźć się nastę-pujące metody publiczne (oprócz konstruktora):

1. dodawanie: `Rational add(Rational arg);`
2. mnożenie: `Rational mul(Rational arg);`
3. odejmowanie: `Rational sub(Rational arg);`
4. dzielenie: `Rational div(Rational arg);`
5. równość: `boolean equals(Rational arg);`
6. porównanie: `int compareTo(Rational arg);`
7. tekstowa reprezentacja liczby: `String toString().`

Metody 1–4 powinny zwracać jako rezultat referencję do nowego obiektu klasy `Rational`, będącego wynikiem operacji wykonanej na argumencie `arg` i `this`. Metoda 5. ma porównywać obiekty klasy `Rational` na podstawie wartości liczb, np. 1/2 = 2/4. Metoda 6. ma działać podobnie, jak odpowiadająca jej metoda `compareTo(Object o)` z interfejsu

`java.lang.Comparable`:

- Jeśli `this` jest równe `arg`, to zwraca **0**.
- Jeśli `this` jest mniejsze od `arg`, to zwraca `-1`.
- Jeśli this jest większe niż arg, to zwraca 1.

Metoda 7. ma zwracać łańcuch znakowy opisujący ten obiekt. Na przykład może to być napis postaci 1/2 lub -1/1.

Zadanie 2.2.

Napisz klasę opisującą równanie kwadratowe o postaci $y = ax^2 + bx + c$. Współczynniki a, b i c powinny być prywatne. Zdefiniuj następujące publiczne funkcje składowe:

- nadającą wartości współczynnikom,
- obliczającą y dla podanego x,
- wyznaczającą liczbę pierwiastków.

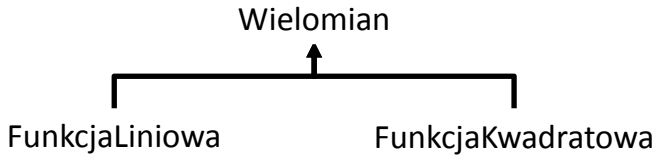
Potrzebne wzory:

- delta: d = $b^2 - 4ac$,
- liczba pierwiastków:

  p =0 : d < 0, 1 : d = 0, 2 : d > 0.

Zadanie 2.3.

Zdefiniuj poniższą hierarchię klas:



tak, aby w wyniku wykonania programu:

```
public class Zadanie {
      public static void main(String[] args) {
      Wielomian w[] = new Wielomian[3];
      w[0] = new FunkcjaLiniowa(2, 1); // 2x + 1
      w[1] = new FunkcjaKwadratowa(1, -2, 2); // x*x - 2x + 2
      w[2] = new FunkcjaKwadratowa(1, 0, -1); // x*x - 1
      for (int i=0; i<3; i++) {
            w[i].wypiszMiejscaZerowe();
            }
      }
}
```

na ekranie pojawił się wynik:

```
-0.5
brak
-1 1
```

Wskazówka

- Wielomian może być klasą abstrakcyjną lub nawet interfejsem.

Zadanie 3.1

Poniższy program:

```
class Kolejka {
      static final int N = 5;
      private Object[] tab;
      private int pocz, zaost, lbel;
            public Kolejka() {
```

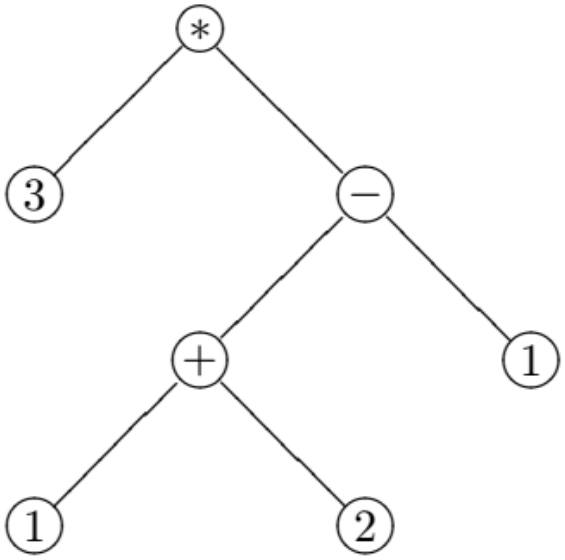
```
            pocz=0; zaost=0; lbel=0;
            tab = new Object[N];
      }

      void doKolejki(Object el) {
            tab[zaost] = el;
            zaost = (zaost+1) % N;
            ++lbel;
      }

      Object zKolejki() {
            int ind = pocz;
            pocz = (pocz+1) % N;
            --lbel;
            return tab[ind];
      }
}

public class Zadanie {
      public static void main(String[] args) {
            Kolejka k = new Kolejka();
            k.doKolejki(new Integer(7));
            k.doKolejki(new String("Ala ma kota"));
            k.doKolejki(new Double(3.14));
            for (int i=1; i<=3; ++i)
                  System.out.println((k.zKolejki()).toString());
      }
}
```

zmodyfikuj tak, aby w funkcji main można było przechwytywać wyjątki przepełnienia (próba dodania, gdy liczba elementów w kolejce wynosi N) i niedomiaru kolejki (próba pobrania elementu kolejki, gdy liczba elementów w kolejce wynosi 0):

```
      public static void main(String[] args) {
            Kolejka k = new Kolejka();
            try {
                  k.doKolejki(new Integer(7));
                  k.doKolejki(new String("Ala ma kota"));
                  k.doKolejki(new Double(3.14));
                  for (int i=1; i<=4; ++i)
                  System.out.println((k.zKolejki()).toString());
            }
                  catch (Przepelnienie e) {
                  System.out.println("Przepełniona kolejka!");
            }
                  catch (Niedomiar e) {
                  System.out.println("Pusta kolejka!");
            }
      }
```

Zadanie 3.2.

Wiadomo, że wyrażenie arytmetyczne może być reprezentowane za pomocą drzewa binarnego. Wierzchołki wewnętrzne takiego drzewa reprezentują działanie, natomiast liście reprezentują stałe. Na przykład wyrażeniu $(3 * ((1 + 2) - 1))$ odpowiada następujące drzewo:

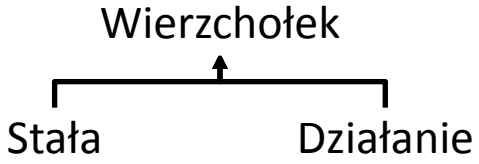


W poniższym programie:

```
abstract class Wierzcholek {
      Wierzcholek lewy, prawy;
      public abstract int wartosc();
}
class Stala extends Wierzcholek {
      private int wart;
      public Stala(int x) {
            wart = x;
      }
      public int wartosc() {
            return wart;
      }
}
class Dzialanie extends Wierzcholek {
      private char op; // operator +, -, / lub *
      public Dzialanie(char znak) {
            op = znak;
      }
      public void dodajLewyArg(Wierzcholek arg) {
            lewy = arg;
      }
      public void dodajPrawyArg(Wierzcholek arg) {
            prawy = arg;
      }
      public int wartosc() {
            switch (op) {
                  case '+': return lewy.wartosc() + prawy.wartosc();
                  case '-': return lewy.wartosc() - prawy.wartosc();
                  case '/': return lewy.wartosc() / prawy.wartosc();
                  case '*': return lewy.wartosc() * prawy.wartosc();
```

```
            }
            return 0;
      }
}
class Wyrazenie {
      private Wierzcholek korzen;
      private Wierzcholek utworzDrzewo(String w, int p, int q) {
            if (p == q)
                  return new Stala(Character.digit(w.charAt(p), 10));
            else {
                  int i = p+1, nawiasy = 0;
                  while ( (nawiasy != 0) || (w.charAt(i) == '(') ||
                  (w.charAt(i) == ')') || (Character.isDigit(w.charAt(i))))
                  {
                        if (w.charAt(i) == '(') ++nawiasy;
                        if (w.charAt(i) == ')') --nawiasy;
                        ++i;
                  }
                  Dzialanie nowy = new Dzialanie(w.charAt(i));
                  nowy.dodajLewyArg(utworzDrzewo(w, p+1, i-1));
                  nowy.dodajPrawyArg(utworzDrzewo(w, i+1, q-1));
                  return nowy;
            }
      }
      public Wyrazenie(String w) {
            korzen = utworzDrzewo(w, 0, w.length()-1);
      }
      public int oblicz() {
            return korzen.wartosc();
      }
}
public class Zadanie {
            public static void main(String[] args) {
            Wyrazenie wyr = new Wyrazenie("(3*((1+2)-1))");
            System.out.println("" + wyr.oblicz());
      }
}
```

stworzono hierarchię klas:



a następnie zaimplementowano klasę `Wyrazenie`, której metoda `oblicz()` zwraca wartość podanego wyrażenia. Zakładamy, że konstruktor akceptuje wyrażenia arytmetyczne skonstruowane zgodnie z gramatyką:

```
<wyrażenie> ::= (<wyrażenie><działanie><wyrażenie>)
<wyrażenie> ::= <stała>
<działanie> ::= + | - | / | *
<stała> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Uzupełnij przytoczony program o obsługę następujących wyjątków:

- dzielenie przez zero,
- niepoprawnie skonstruowane wyrażenie.

Zadanie 4.1.

Napisz prosty edytor tekstowy, w którym będzie możliwość zapisywania tekstu do pliku w jednym z wybranych standardów kodowania znaków: UTF-8, ISO-8859-2 lub windows-1250.

Wskazówki:

- Skorzystaj z klas OutputStreamWriter oraz FileOutputStream.
- Łańcuchami reprezentującymi wymienione standardy kodowania znaków są: UTF8, ISO8859 2 oraz Cp1250.

Zadanie 4.2.

Napisz program kompresujący plik do formatu GZIP oraz program rozpakowujący plik GZIP.

Wskazówka

- Skorzystaj z klas GZIPOutputStream oraz GZIPInputStream.

Zadanie 5.1.

Dostosuj poniższą klasę:

```
class Wspolrzedna {
      private int x, y;
      public Wspolrzedna(int _x, int _y) {
            x = _x;
            y = _y;
      }
      public String toString() {
            return "(" + x + ", " + y + ")";
      }
}
```

do wymagań stawianych wobec elementów dodawanych do kontenera `TreeSet`, tak aby w wyniku wykonania programu:

```
import java.util.*;
//Tu wstaw zmodyfikowaną klasę przechowującą współrzędne punktu
public class Zadanie {
      private static void wypiszElementy(TreeSet zbior) {
            Iterator it = zbior.iterator();
            while (it.hasNext()) {
                  System.out.println((it.next()).toString());
            }
      }
      public static void main(String[] args) {
            TreeSet zbior = new TreeSet();
            zbior.add( new Wspolrzedna(2, 3) );
            zbior.add( new Wspolrzedna(-3, 0) );
            zbior.add( new Wspolrzedna(-1, 2) );
            zbior.add( new Wspolrzedna(-1, 2) );
            zbior.add( new Wspolrzedna(-3, -2) );
            wypiszElementy(zbior);
      }
}
```

punkty zbioru `TreeSet` były wyświetlone na ekranie w kolejności leksykograficznej (czyli (−3, −2), (−3, 0), (−1, 2), (2, 3)).

Wskazówka

- Klasa `Wspolrzedna` powinna implementować interfejs `Comparable`.

Zadanie 5.2.

Dostosuj klasę `Wspolrzedna` z poprzedniego zadania do wymagań stawianych wobec elementów dodawanych do kontenera `HashMap`. Wówczas w wyniku wykonania programu:

```
import java.util.*;
//Tu wstaw zmodyfikowaną klasę przechowującą współrzędne punktu
public class Zadanie {
      public static void main(String[] args) {
      HashMap mapa = new HashMap();
      mapa.put(new Wspolrzedna(2, 3), new String("czerwony"));
      mapa.put(new Wspolrzedna(-3, 0), new String("czarny"));
      mapa.put(new Wspolrzedna(-1, 2), new String("czerwony"));
      mapa.put(new Wspolrzedna(2, -1), new String("czarny"));
      Wspolrzedna w = new Wspolrzedna(-1, 2);
      System.out.println("Punkt " + w.toString()
      + " ma kolor " + mapa.get(w));
      }
}
```

na ekranie zostanie wyświetlony tekst:

```
Punkt (-1, 2) ma kolor czerwony
```

Wskazówka

- Klasa `Wspolrzedna` powinna przesłonić metody `hashCode` oraz `equals`.

Zadanie 5.3

W poniższej klasie Graf:

```
import java.util.*;
class Graf {
      private int n; // liczba wierzchołków, V = {0,1,...,n-1}
      private LinkedList[] tab; // tablica wierzchołków połączo-
                                // nych z danym wierzcholkiem
      public Graf(String lan) {
            StringTokenizer st = new StringTokenizer(lan, "() ,");
            n = Integer.parseInt(st.nextToken());
            tab = new LinkedList[n];
            for (int i=0; i<n; ++i)
            tab[i] = new LinkedList();
            while (st.hasMoreTokens()) {
                  tab[Integer.parseInt(st.nextToken())].add(
                  new Integer(st.nextToken()));
            }
      }
      public String toString() {
      ...
      }
}
public class Zadanie {
      public static void main(String[] args) {
            Graf g = new Graf("4, (0,1), (1,2), (3,0), (1,3)");
            System.out.println(g.toString());
      }
}
```

zdefiniuj metodę `toString` w taki sposób, aby graf był przedstawiany jako tablica ciągów wierzchołków połączonych z kolejnymi wierzchołkami grafu skierowanego:

```
0: 1
1: 2 3
2:
3: 0
```

Wskazówki

• W celu wielokrotnego dołączania łańcucha (lub liczby) na końcu innego łańcucha najlepiej skorzystać z klasy **`StringBuffer`** i jej metody **`append`**.

- Przejście do nowego wiersza realizujemy dołączając do łańcucha sekwencję sterującą "\n".

Zadanie 6.1

W poniższym programie użytkownik ma możliwość wprowadzania tekstu do okienka. Działający w programie wątek zamienia we wpisywanym tekście wystąpienie znaku klamry otwierającej na słowo **`begin`** oraz znaku klamry zamykającej na słowo **`end`**.

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

class Zamieniacz extends Thread {
      JTextArea okno;
      volatile boolean zakonczyc;
      public Zamieniacz(JTextArea comp) {
            okno = comp;
            zakonczyc = false;
      }
      public void run() {
            while (! zakonczyc) {
            try {
                  String tekst = okno.getText();
                  int indeks = tekst.indexOf("{");
                  if (indeks >= 0) {
                        okno.replaceRange("begin", indeks, indeks+1);
                        okno.setCaretPosition(tekst.length()+4);
                  }
                  else {
                        indeks = tekst.indexOf("}");
                        if (indeks >=0) {
                              okno.replaceRange("end", indeks,
                              indeks+1);
                              okno.setCaretPosition(tekst.length()+2);
                        }
                  }
                  sleep(2000);
            }
            catch (Exception e) {}
            }
      }
}

public class NewJFrame extends JFrame {
      public NewJFrame() {
            initComponents();
            setSize(350, 250);
            watek = new Zamieniacz(jTextArea1);
            watek.start();
      }
      private void initComponents() {
      jScrollPane1 = new JScrollPane();
      jTextArea1 = new JTextArea();
      setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
      addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent evt) {
                  formWindowClosing(evt);
            }
            });
            jTextArea1.setPreferredSize(new Dimension(300, 200));
            jScrollPane1.setViewportView(jTextArea1);
            getContentPane().add(jScrollPane1, BorderLayout.CENTER);
            pack();
      }
      private void formWindowClosing(WindowEvent evt) {
            watek.zakonczyc = true;
            watek = null;
      }
      public static void main(String args[]) {
```

```
            EventQueue.invokeLater(new Runnable() {
                  public void run() {
                  new NewJFrame().setVisible(true);
                  }
            });
      }
      private JScrollPane jScrollPane1;
      private JTextArea jTextArea1;
      private Zamieniacz watek;
}
```

Napisz oraz dodaj do programu wątek sprawdzający co 10 sekund, czy użytkownik wprowadził do okienka tekstowego słowo niecenzuralne (np. *„cholera”*) i informujący o tym fakcie za pomocą odpowiedniego komunikatu przekazanego do metody **`JOptionPane.showMessageDialog()`.**

Zadanie 6.2.

W poniższym programie zdefiniowano klasę **`KolejkaKomunikatow`**, do której może odwoływać się kilka wątków naraz.

```
import java.util.*;
class KolejkaKomunikatow {
      Vector kolejka = new Vector();
      public synchronized void wyslij(Object ob) {
            kolejka.addElement(ob);
      }
      public synchronized Object odbierz() {
            if (kolejka.size() == 0) return null;
            Object ob = kolejka.firstElement();
            kolejka.removeElementAt(0);
            return ob;
      }
}
class Watek extends Thread {
      private KolejkaKomunikatow koko;
      private int istart;
      public Watek(KolejkaKomunikatow kk, int pocz) {
            koko = kk;
            istart = pocz;
      }
      public void run() {
            try {
                  for (int i=istart; i<=10; i+=2) {
                        koko.wyslij(new Integer(i));
                        Thread.sleep(50);
                  }
            }
            catch (InterruptedException e) {};
      }
}
public class Zadanie {
      public static void main(String args[]) {
            KolejkaKomunikatow k = new KolejkaKomunikatow();
            Watek w1 = new Watek(k, 1);
            Watek w2 = new Watek(k, 2);
```

```
            w1.start();
            w2.start();
            try {
                  w1.join();
                  14 Zbiór zadań do przedmiotu programowanie
                  obiektowe
                  w2.join();
            }
            catch (InterruptedException e) {};
            Object ob = k.odbierz();
            while (k != null) {
                  System.out.println(((Integer)ob).toString());
                  ob = k.odbierz();
            }
      }
}
```

Zaimplementuj w podobny sposób odwoływanie się przez wątki do klasy **`HashMap`** (klucz może być obiektem klasy **`String`**, a wartość obiektem klasy **`Integer`**).

Zadanie 7.1

Poniższy program łączy się z podanym (jako parametr wywołania) „daytime" serwerem na porcie 13, a następnie odczytuje komunikat wysyłany przez serwer.

```
import java.net.*;
import java.io.*;
public class Zadanie {
      public static void main(String[] args) {
            String nazwahosta;
                  if (args.length > 0) {
                  nazwahosta = args[0];
            }
            else {
                  nazwahosta = "time-a.nist.gov";
            }
            try {
                  Socket gniazdo = new Socket(nazwahosta, 13);
                  InputStream strumien = gniazdo.getInputStream();
                  BufferedReader bufor = new BufferedReader(
                  new InputStreamReader(strumien));
                  String wiersz = "";
                  while (wiersz != null) {
                        System.out.println(wiersz);
                        wiersz = bufor.readLine();
                  }
            }
            catch (UnknownHostException e) {
                  System.err.println(e);
            }
            catch (IOException e) {
                  System.err.println(e);
            }
      }
}
```

Listę wybranych serwerów podających aktualną datę i czas przedstawiono poniżej:

- time-a.nist.gov
- time-b.nist.gov
- time-nw.nist.gov
- time.windows.com

Po połączeniu się z jednym z nich przez port 37 wysyła on 32 bity reprezentujące liczbę sekund, które upłynęły od północy 1 stycznia 1900 r. Napisz program odczytujący tę liczbę.

Wskazówki

- Skorzystaj z metody **`read()`** klasy **`InputStream`**.
- Cztery bajty zamień na liczbę typu long za pomocą operatorów **`<<`** oraz **`|`**.

Zadanie 7.2

Rozważmy następujący program-serwer.

```
import java.net.*;
import java.io.*;

class Gracz implements Runnable {
    private int plansza[][];
    // liczba >= 100 to mina, 0, 1, ..., 8 -- ile wokól min
    private boolean klikniete[][];
    private PrintWriter out;
    private BufferedReader in;
    private Socket polaczenie;
    private void InicjujPlansze() {
        plansza = new int[11][11];
        klikniete = new boolean[11][11];
        int w, k, licznik;
        licznik = 0;
        while (licznik < 10) {
        w = (int)(Math.random()*9) + 1;
        k = (int)(Math.random()*9) + 1;
        if (plansza[w][k] < 100) {
        ++licznik;
        plansza[w][k] = 100;
        ++plansza[w-1][k-1];
        ++plansza[w-1][k];
        ++plansza[w-1][k+1];
        ++plansza[w][k-1];
        ++plansza[w][k+1];
        ++plansza[w+1][k-1];
        ++plansza[w+1][k];
        ++plansza[w+1][k+1];
        }
        }
    }
    public Gracz(Socket polaczenie) {
        InicjujPlansze();
```

```
    this.polaczenie = polaczenie;
    try {
    out = new PrintWriter(polaczenie.getOutputStream(), true);
    in = new BufferedReader(
    new InputStreamReader(polaczenie.getInputStream()));
    }
    catch (IOException e) {
    System.out.println(e.toString());
    }
}
public void run() {
    int w, k, odkryte, liczba;
    String wsp, odp;
    boolean koniec = false;
    odkryte=0;
    try {
        out.println("OK.");
        while ((!koniec) && (odkryte<71)) {
            wsp = in.readLine();
            if (wsp == null) koniec = true;
            else {
                try {
                    liczba = Integer.parseInt(wsp);
                    w = (int)((liczba-1)/9) + 1;
                    k = (liczba-1) % 9 + 1;
                }
                catch (NumberFormatException e) {
                    w = 200;
                    k = 200;
                }
                if ((w>=1) && (w<=9) && (k>=1) && (k<=9)) {
                    if (plansza[w][k] >= 100) {
                        out.println("bum");
                        koniec = true;
                    }
                    else {
                        out.println(Integer.toString(plansza[w][k]));
                        if (!klikniete[w][k]) ++odkryte;
                    }
                    klikniete[w][k] = true;
                }
            }
        }
    }
    catch (IOException e) {
        System.out.println(e.toString());
    }
    finally {
        try {
        polaczenie.close();
        }
        catch (IOException e) {}
        }
    }
}

public class Serwer {
    public static void main(String[] args) {
    ServerSocket server;
    try {
    server = new ServerSocket(9696);
```

```
            while(true) {
            Socket polaczenie = server.accept();
            Thread t = new Thread(
            new Gracz(polaczenie));
            t.start();
            }
            }
            catch (IOException e) {
            System.out.println(e.toString());
            }
        }
    }
```

Jest to program, który dla każdego klienta, który się z nim połączy generuje pole minowe znane z windowsowej gry „Saper". Numerację pól tego pola przedstawiono na poniższym rysunku:

```
 1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36
37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72
73 74 75 76 77 78 79 80 81
```

Serwer na tym polu rozmieszcza losowo 10 min. Na każdym polu wolnym od miny umieszcza liczbę od 0 do 8 określającą ile min jest wokół niego. Napisz klienta, który:

1. połączy się z serwerem przez port 9696;
2. odbierze wiersz tekstu (słowo „OK.");
3. w pętli, do momentu „odkrycia" wszystkich 71 wolnych pól lub natrafienia na minę (serwer wtedy odpowiada „bum"), będzie przesyłał serwerowi liczbę x (jako String) podaną przez użytkownika i odbierał od serwera liczbę (również jako String) określającą ile jest min wokół pola x.

Zawartość