Język programowania JAVA

Vincent Van GOGH: *Mężczyzna pijący filiżankę kawy*

Klasy

Java jest językiem zorientowanym obiektowo. Wszystkie elementy opisujące dane, za wyjątkiem zmiennych prostych są obiektami. Program też jest obiektem pewnej klasy. W Javie podstawowym elementem jest **klasa**. Każdy program Javy składa się z jednej lub więcej klas i nie zawiera poza nimi żadnych innych elementów.

Tworzenie klas

- Aby utworzyć obiekt należy najpierw zdefiniować klasę, która jest wzorem dla tworzenia obiektów tej klasy.
- Klasa definiuje zarówno dane jak i algorytmy służące do ich przetwarzania. Dane sa zapisywane w klasie w postaci pól (zmiennych lub stałych składowych), a algorytmy w postaci metod.
- Metoda jest wydzielonym fragmentem programu, na ogół zawierający parametry, który może operować na ściśle określonych danych.
- Aby utworzyc klasę, należy użyc słowa kluczowego **class** nazwę klasy i pary nawiasów { }.
- Wewnątrz nawiasów klamrowych umieszczone są zmienne egzemplarzowe i metody.

Tworzenie klas – składnia

```
1  class [atrybuty dostępu] nazwa_klasy extends nazwa_nadklasy
2  {
3    typ nazwa_zmiennej_egzemplarzowej1;
4    typ nazwa_zmiennej_egzemplarzowej2;
5    ...
6    typ nazwa_zmiennej_egzemplarzowejN;
7    typ nazwa_metody1(lista_parametrów)
8    {
9      część_główna_metody
10   }
11     typ nazwa_metody2(lista_parametrów)
12   {
13     część_główna_metody
14   }
15 ...
16  typ nazwa_metodyN(lista_parametrów)
17   {
18     część_główna_metody
19   }
20 }
```

Atrybuty dostępu – modyfikatory

Atrybut dostępu, inaczej modyfikator dostępu określa w jaki sposób inne obiekty mogą otrzymać dostęp do danego pola, metody, definicji klasy czy interfejsu. Wyróżnia sie cztery atrybuty dostepu:

- **public** (publiczny) – możliwość dostępu z dowolnej klasy,
- **protected** (chroniony) – możliwość dostępu z klasy pochodnej lub należącej do tego samego pakietu,
- **private** (prywatny) – możliwość dostępu wyłącznie w danej klasie,
- dostęp domyślny (bez podanego specyfikatora dostepu) – możliwość dostępu wyłącznie z klas należących do tego samego pakietu.

Inne modyfikatory – cd

- **final** – pole obiektu oznaczone jako final po inicjacji nie może być modyfikowane.
- **static** – modyfikator static oznacza iż pole obiektu ma taką samą wartość dla wszystkich obiektów danej klasy. Formalnie oznacza to iż wszystkie obiekty danej klasy odwołują się do tego samego miejsca w pamięci. Jeżeli metoda jest oznaczona jako statyczna to może być wywołana bez potrzeby tworzenia obiektu klasy definiującej tą metodę.

Modyfikator final

Modyfikator **final** ma trzy zastosowania:

- umożliwia definiowanie klas, które nie mogą być dziedziczone
- pozwala tworzyć metody, których nie można przykrywać w podklasach
- umożliwia deklarowanie zmiennych, których wartości nie można zmieniać (odpowiedniki stałych z innych języków programowania). Elementy zadeklarowane ze słowem kluczowym **final** są to elementy finalne. Piszemy je dużymi literami.

```
1 final int ILOSC = 100;
2 final float PI = 3.14159;
```

Modyfikator abstract

Modyfikator **abstract** występujący przy deklaracji klasie oznacza, że mamy do czynienia z definicją, która jest nie kompletna lub też powinna być za taką uważana. Tylko klasa tak oznaczona może zawierać metody, które nie posiadają implementacji. Dana klasa ma metody abstrakcyjne, jeżeli:

- w klasie jest umieszczona jakakolwiek metoda oznaczona jako **abstract**
- jakakolwiek klasa nadrzędna posiada metody abstrakcyjne, które nie zostają implementowane
- klasa dziedziczy pośrednio lub bezpośrednio interfejs i nie implementuje wszystkich jego metod

Klasa abstrakcyjną jest używana w celu uogólnienia zbioru innych klas (zawiera pewne wspólne właściwości tych klas). Może także część tych właściwości definiować, pozwala natomiast zdefiniować interfejs dostępu do innych właściwości, których implementacja musi już nastąpić w którejś klasie potomnej.

Inne modyfikatory

- **strictfp** – Metoda lub klasa oznaczona w ten sposób będzie wykonywana tak by wszystkie obliczenia były zgodne ze standardem IEEE-754.
- **native** – Zaawansowany modyfikator, którym oznaczane są metody wykonywane przez JNI (Java Native Interface). W praktyce oznacza to, że metody te są implementowane w języku innym niż Java, a JVM wywołując je odwołuje się do mechanizmów pozwalających na komunikację z np C++.
- **transisten** – Zaawansowany modyfikator związany z JPA (Java Persistence Api). Pola oznaczone w ten sposób nie są utrwalane.
- **volatile** – Zaawansowany modyfikator związany z wielowątkowością. Zmienna nim oznaczona może być modyfikowana przez wiele wątków jednocześnie.

Zmienne egzemplarzowe

- Enkapsulację danych uzyskuje się przez deklarowanie **zmiennych egzemplarzowych** klas.
- Deklaracje te muszą znajdować się w obrębie danej klasy.
- Zmienne egzemplarzowe mogą być dowolnego typu, może to być jeden z typów prostych lub jeden z typów kasy.
- Przykład:

```
1  class Punkt
2  {
3    int x, y;
4  }
```

Podana klasa deklaruje dwie zmienne int x i y.

Definiowanie metod

Metody, czyli podprogramy związane są z poszczególnymi klasami, tworzą ich interfejsy. Implementacje wszystkich metod danej klasy muszą znajdować się w obrębie jej definicji, na tym samym poziomie co zmienne egzemplarzowe. Metody mogą korzystać ze zmiennych egzemplarzowych swojej klasy w taki sposób jakby były ich zmiennymi lokalnymi.

Definiowanie metod

Metody, czyli podprogramy związane są z poszczególnymi klasami, tworzą ich interfejsy. Implementacje wszystkich metod danej klasy muszą znajdować się w obrębie jej definicji, na tym samym poziomie co zmienne egzemplarzowe. Metody mogą korzystać ze zmiennych egzemplarzowych swojej klasy w taki sposób jakby były ich zmiennymi lokalnymi.

- Deklaracja metody musi określać typ zwracanej przez nią wartości, a ponadto może zawierać listę parametrów formalnych. Jej składnia:

```
1  typ_powrotny nazwa_metody (lista_parametrów_formalnych)
2  {
3    część_główna_metody
4  }
```

Definiowanie metod

Metody, czyli podprogramy związane są z poszczególnymi klasami, tworzą ich interfejsy. Implementacje wszystkich metod danej klasy muszą znajdować się w obrębie jej definicji, na tym samym poziomie co zmienne egzemplarzowe. Metody mogą korzystać ze zmiennych egzemplarzowych swojej klasy w taki sposób jakby były ich zmiennymi lokalnymi.

- Deklaracja metody musi określać typ zwracanej przez nią wartości, a ponadto może zawierać listę parametrów formalnych. Jej składnia:

```
1  typ_powrotny nazwa_metody (lista_parametrów_formalnych)
2  {
3    część_główna_metody
4  }
```

- Jeżeli metoda nie zwraca żadnej wartości - **void**

Definiowanie metod

Definiowanie metod

- Nazwy metod i zmiennych egzemplarzowych nie mogą się powtarzać w obrębie wspólnej klasy.

Definiowanie metod

- Nazwy metod i zmiennych egzemplarzowych nie mogą się powtarzać w obrębie wspólnej klasy.
- Lista parametrów formalnych – pary identyfikatorów: typ i nazwa parametru.

Definiowanie metod

- Nazwy metod i zmiennych egzemplarzowych nie mogą się powtarzać w obrębie wspólnej klasy.
- Lista parametrów formalnych – pary identyfikatorów: typ i nazwa parametru.
- Jeżeli metoda nie pobiera żadnych parametrów – to tylko same nawiasy ().

Definiowanie metod

- Nazwy metod i zmiennych egzemplarzowych nie mogą się powtarzać w obrębie wspólnej klasy.
- Lista parametrów formalnych – pary identyfikatorów: typ i nazwa parametru.
- Jeżeli metoda nie pobiera żadnych parametrów – to tylko same nawiasy ().
- Przykład klasy z definicją metody:

```
1  class Punkt
2  {
3    int x, y;
4    void init(int a, int b)
5    {
6      x = a;
7      y = b;
8    }
9  }
```

Obiekt – operator new

Obiekt jest to naturalna reprezentacja klasy. Przechowuje informacje o zmiennych i pozwala się do nich odwoływać, a także za jego pomocą możemy wywoływać metody. Utworzenie nowego obiektu – czyli instancji klasy dokonujemy za pomocą słowa kluczowego **new**. Operator ten:

- rezerwuje pamięć wymaganą przez obiekt.
- zwraca adres, za pomocą którego można się do tego obiektu odwołać.

```
punkt pkt = new();
```

zmienna pkt została zadeklarowana jako zmienna typu klasa, nie jest obiektem, lecz odwołaniem do niego.

Operator kropka

Dostęp do zmiennych i metod obiektów uzyskuje się za pomocą operatora oznaczonego kropka •. Składnia wyrażenia uzyskującego dostęp do zmiennej:

```
odwołanie_do_obiektu.nazwa_zmiennej
```

- Zmienna znajdująca się z lewej strony operatora • musi być odwołaniem do obiektu utworzonego za pomocą operatora **new**
- Identyfikator z prawej strony musi być nazwą zmiennej egzemplarzowej.

```
1 pkt.x = 5;
2 pkt.x = 3;
```

Wywołanie metody

Operator • umożliwia nie tylko uzyskanie dostępu do zmiennych egzemplarzowych klas, ale także wywołanie ich metod. Składnia wywołania metody:

```
odwołanie_do_obiektu.nazwa_metody(lista_argumentów)
```

- Zmienna znajdująca się z lewej strony operatora • musi być odwołaniem do obiektu utworzonego za pomocą operatora **new**.
- Identyfikator z prawej strony musi być nazwą jednej z jego metod.
- lista argumentów – lista wartości, których liczba i typy muszą być zgodne z typami wywołanej metody.

```
1 Punkt pkt = new punkt();
2 pkt.init(3, 7);
```

this

Słowo kluczowe **this**, umożliwia uzyskanie dostępu do zmiennych egzemplarzowych i metod bieżacej klasy. Innymi słowy jest odwołaniem do bieżącej klasy.

```
1  class Punkt
2  {
3    int x,y;
4    void init(int a, int b)
5    {
6      this.x = a;
7      this.y = b;
8    }
9  }
```

Konstruktory

Konstruktor jest to metoda, która umożliwia automatyczną inicjalizację zmiennych egzemplarzowych klasy podczas deklarowania jej obiektów. Aby taką klasę utworzyć, należy umieścić wszystkie instrukcje inicjalizujące w jednej metodzie o takiej samej nazwie jak nazwa klasy. Jest ona wywoływana automatycznie podczas tworzenia nowego obiektu za pomocą operatora **new**. Przykład:

```
1  class Punkt
2  {
3    int x, y;
4    Punkt(int a, int b)
5    {
6      x = a;
7      y = b;
8    }
9  }
```

Metoda ta nie ma typu powrotnego

Przeładowywanie metod

Mechanizm przeładowywania pozwala na tworzenie metod o takich samych nazwach, ale różnych parametrach. Nieprawidłowe jest utworzenie w jednej klasie dwóch metod o identycznej nazwie i przyjmującej takie same parametry, a także metody o takiej samej nazwie i parametrach, ale różniące się tylko zwracanym typem.

```
1  class Test
2  {
3    int dodaj(int a, int b){
4      return a+b;
5    }
6
7    double dodaj(double a, double b){
8      return a+b;
9    }
10 }
```

Dziedziczenie

Dziedziczenie to podstawowy mechanizm programowania obiektowego. Dzięki niemu możemy utworzyć spójną i łatwą do zrozumienia hierarchię klas

Dziedziczenie

Dziedziczenie to podstawowy mechanizm programowania obiektowego. Dzięki niemu możemy utworzyć spójną i łatwą do zrozumienia hierarchię klas

- **Podklasa** danej **nadklasy** (superklasa) dziedziczy wszystkie jej zmienne egzemplarzowe i metody.

Dziedziczenie

Dziedziczenie to podstawowy mechanizm programowania obiektowego. Dzięki niemu możemy utworzyć spójną i łatwą do zrozumienia hierarchię klas

- **Podklasa** danej **nadklasy** (superklasa) dziedziczy wszystkie jej zmienne egzemplarzowe i metody.
- Podklasa definiuje własne elementy.

Dziedziczenie

Dziedziczenie to podstawowy mechanizm programowania obiektowego. Dzięki niemu możemy utworzyć spójną i łatwą do zrozumienia hierarchię klas

- **Podklasa** danej **nadklasy** (superklasa) dziedziczy wszystkie jej zmienne egzemplarzowe i metody.
- Podklasa definiuje własne elementy.
- Aby rozszerzyć jakąś klasę należy użyć słowa kluczowego extends w nagłówku klasy:

```
1  class Punkt3D extends Punkt{
2    int z;
3    Punkt3D(int x, int y, int z){
4        this.x = x;
5        this.y = y;
6        this.z = z;
7    }
8    Punkt3D(){
9      this(2, 0, -1);
10   }
11 }
```

super

Słowo kluczowe **this** umożliwia korzystanie ze zmiennych i konstruktora bieżącej klasy. Słowo **super** pozwala uzyskać dostęp do konstruktora i także każdej innej metody nadklasy. W ostatnim przykładzie konstruktor klasy Punkt3D zawiera dwie instrukcje, ktore występują jako jedyne w konstruktorze jej nadklasy Punkt:

```
1       this.x = x;
2       this.y = y;
```

Możemy za pomocą słowa **super** wywołac konstruktor klasy Punkt:

```
1
2       super(x, y)
3       this.z = z;
```

super – przykład

Metoda main() klasy Punkt3Ddist tworzy obiekt klasy Punkt3, inicjalizuje jego wspólrzędne i wyświetla na ekranie.

```
1  class Punkt3D extends Punkt{
2    int z;
3    Punkt3D(int x, int y, int z){
4       super(x, y);
5       this.z = z;
6    }
7    Punkt3D(){
8      this(2, 0, -1);
9    }
10 }
11 class Punkt3Ddist
12 {
13   public static void main(String arg[])
14   {
15   Punkt3D pkt = new Punkt3D(3,5,6);
16   System.out.println("x = " + pkt.x + ", y = " + pkt.y +", z = " + pkt.z);
17   }
18 }
```

Przykrywanie metod

Ponowne definiowanie metody odziedziczonj z nadklasy nazywamy **przykrywaniem**.Metody te różnią się sygnaturą typów.

Finalizacja

Finalizacja to zwalnianie pamięci. Interpreter Javy automatycznie czyści pamięć nie ma więc potrzeby samemu zwalniać przydzielonych bloków pamięci przez instrucję np. delate znanych z języka C++. Jeżeli klasa korzysta z zasobów nie należących do Javy (np. pliki), wówczas w danej klasie należy zadeklarować metody o nazwie **finalize()** i umieścić w niej wszystkie instrukcje potrzebne do zwolnienia zasobów wewnętrznych.

Pobieranie danych od użytkownika

Do pobierania informacji służy strumień System.in. Jest z nim związana klasa Scanner. Oto prosty program do pobierania danych od użytkownika i wyświetlający powitanie.

```
1  import java.util.Scanner;
2
3  public class Witaj{
4    public static void main(String[] args){
5            String nick;
6            Scanner wejscie = new Scanner(System.in);
7
8            nick = wejscie.nextLine();
9
10           System.out.println("Witaj␣"+nick);
11   }
12 }
```

wejscie – obiekt do odebrania danych od użytkownika

Wczytywanie danych

1. W nagłówku importujemy klasę Scanner.
2. Deklarujemy zmienną **nick** do zapisania pobranych danych.
3. Następnie tworzymy nowy obiekt Scanner w dwóch etapach:
   - deklaracja

```
1 Scanner wejscie
```

   - utworzenie obiektu za pomocą operatora new:

```
1 wejscie = new Scanner(System.in)
```

   - Jako parametr konstruktora obiektu Scanner podajemy strumień wejścia System.in

     Te dwie instrukcje można zapisać w jednej lini

```
1 Scanner wejscie = new Scanner(System.in);
```

4. imie = wejscie.nextLine() – tą instrukcją odebrana jest od użytkownika jedna linia znaków (tekstu zakończonego klawiszem enter)

Wczytywanie danych

Klasa Scanner oferuje szereg metod do odczytu danych:

- nextInt() — odczytuje kolejną liczbę całkowitą
- nextDouble() — czyta kolejną liczbę zmiennoprzecinkową
- nextLong() – czyta kolejną liczbę typu Long
- next() – zwraca odczytaną następną wartość dowolnego typu
- hasNext() – zwraca wartość True jeśli następna wartość dowolnego typu jest dostępna do odczytu.
- hasNextInt() – zwraca wartość True jeśli następna wartość jest typu Integer.
- hasNextDouble() – zwraca wartość True jeśli następna wartość jest typu Double.

i inne

Wyświetlanie danych

Do wyświetlania danych używamy metody System.out.print(nick). Można w ten sposób wyświetlić zarówno liczby, znaki jak i łańcuchy znaków. Metoda println powoduje wyświetlenie komunikatu i przejście do nowej lini. Istnieje też metoda formatowanego wyświetlania danych.

Zapis i odczyt z plików

Podstawowa klasa, która pozwola utworzyć obiekt przechowujący dane pliku, to File. Tworzymy go w następujący sposób:

```
1  File plik = new File("nazwa_pliku.txt");
```

W ten sposób mamy do dyspozycji obiekt File o nazwie plik, na którym możemy wykonywać operacje:

- zapisywania
- odczytywania.

Odczyt

Na najprostszy odczyt danych pozwala już wcześniej poznana klasa Scanner. Aby utworzyć strumień, należy użyć takiej konstrukcji:

```
1 Scanner wejscie = new Scanner(new File("nazwa_pliku.txt"));
```

Widzimy tutaj typową dla Javy i języków obiektowych konstrukcję (wzorzec projektowy) zwaną dekoratorem, często stosowanym w Javie. Klasa Scanner „obudowuje" klasę File, rozszerzając jej funkcjonalność o możliwość odczytu pliku. Gdy mamy utworzony obiekt Scanner z przekazanym jej odpowiednim obiektem File możemy już używać odpowiednich metod odczytu.

```
1 String text = wejscie.nextLine();
```

Wczytamy w ten sposób do zmiennej text linię tekstu z pliku tekstowego wejście.

Odczyt z pliku – przykład

Tworzymy plik, w którym zapisujemy jedną dowolną linię tekstu, przykładowo „Studia podyplomowe”. Plik ten nazwiemy studia.txt. Kod programu do odczytania tego tekstu:

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.util.Scanner;
4
5  public class odczyt{
6    public static void main(String[] args) throws FileNotFoundException{
7            File plik = new File("studia.txt");
8            Scanner in = new Scanner(plik);
9
10           String tekst = in.nextLine();
11           System.out.println(tekst);
12   }
13 }
```

Plik musi być w tym samym folderze, co plik .class, w innym wypadku należy podać ścieżkę do pliku w miejsce jego nazwy. Dopisek FileNotFoundException – jest to wyjątek, który musimy zgłosić.

Zapis

Kolejnym ważnym zagadnieniem jest zapis do pliku. Możemy tutaj użyć klasy PrintWriter i utworzyć jej obiekt, ale uwaga, jako parametr konstruktora podajemy tutaj tylko nazwę pliku, a nie obiekt File.

```
1  PrintWriter zapis = new PrintWriter("nazwa_pliku.txt");
```

Zapiszemy do pliku studia.txt zdanie „Studia podyplomowe” przy pomocy metody print().

```
1  import java.io.FileNotFoundException;
2  import java.io.PrintWriter;
3
4  public class Zapis{
5    public static void main(String[] args) throws FileNotFoundException{
6            PrintWriter zapis = new PrintWriter("studia.txt");
7            zapis.println("Studia podyplomowe");
8            zapis.close();
9    }
10 }
```

Losowanie

```
1  package losowanie;
2
3  // Importowanie biblioteki potrzebnej do losowania liczb
4  import java.util.Random;
5
6  public class Main {
7
8      public static void main(String[] args) {
9          /*
10          * Przykład losowania liczb
11          */
12
13         // Stworzenie obiektu klasy Random, który posłuży do loswania liczb
14         Random r = new Random();
15
16         /*
17          * Losowanie liczb odbywa się po wywołaniu którejś z metod obiektu r
18          * (klasy Random).
19          * W zależności od typu liczby, którą należy wylosować, trzeba wybrać
20          * odpowiednią metodę np.
21          *      r.nextInt() - wylosuje liczbę całkowitą z zakresu int,
22          *      r.nextFloat() - wylosuje liczbę rzeczywistą z zakresu float,
23          *      itd. lista dostępnych metod klasy Random wyświetli się po
24          *      wpisaniu nazwy obiektu i kropki w tym przypadku: r.
25          *
26          * W przypadku zapisu r.nextInt(n); wylosowana zostanie liczba z
27          * zakresu od 0 do n-1, czyli chcąc wylosować liczbę z zakresu
28          * od 1 do 10 (domkniętego) należy zapisać:
29          * r.nextInt(10)+1;
30          */
31
32         // Losowanie liczby z zakresu [0,10] do zmiennej a.
33         int a = r.nextInt(11); // deklaracja i definicja zmiennej
```

Zaokrąglanie

```
1  package zaokraglanie;
2
3  // Importowanie biblioteki potrzebnej do wczytywania danych
4  import java.util.Scanner;
5
6  public class Main {
7
8      public static void main(String[] args) {
9          /*
10          * Przykład zaokrąglania liczb.
11          */
12         Scanner in = new Scanner(System.in);
13
14         double pi = Math.PI;
15         System.out.println("Całe Math.PI:\n" + pi);
16
17         System.out.println();
18
19         pi = Math.round(Math.PI);
20         System.out.println("Pełne zaokrąglenie, Math.round(Math.PI):\n" + pi);
21
22         System.out.println();
23         // Zaokrąglenie do 2 miejsca po przecinku:
24         pi = Math.PI;
25         pi *= 100; // pi = pi * 100;
26         pi = Math.round(pi);
27         pi /= 100; // pi = po / 100;
28         System.out.println("Zaokrąglenie do 2 miejsca po przecinku:\n" + pi);
29
30         System.out.println();
31         // Zaokrąglenie do 5 miejsca po przecinku:
32         pi = Math.PI;
33         pi *= 100000; // pi = pi * 100;
```