Język JAVA
podstawy programowania

Jacek Rumiński

Wykład 4, część 3

Plan wykładu:

1. Podstawy modelowania obiektowego
2. Konstruktory
3. Dziedziczenie, związki pomiędzy klasami, UML
4. Polimorfizm
5. Klasy abstrakcyjne i interfejsy
6. Adaptery, klasy anonimowe
7. Tablice, kolekcje i typ wyliczeniowy

Adapter

Implementacja interfejsu wymaga stworzenie definicji dla wszystkich metod abstrakcyjnych (jeśli nie – otrzymamy klasę abstrakcyjną).

Częstym przypadkiem może być sytuacja, w której projektant interfejsu przewidział wiele funkcji dla modelowanego elementu (np. radia, samochodu), ale dana implementacja (wykonawca elementu) pragnie obsługiwać jedynie część tych funkcji (np. ograniczenia ekonomiczne). Co wtedy?

Typowym rozwiązaniem jest implementacja tych abstrakcyjnych metod, których funkcjonalność nas interesuje, a pozostałe implementujemy jako puste – bez instrukcji.

Rozważmy przykład obsługi baz danych. Grupa przedstawicieli firm wypracowuje standard interfejsu dostępu do baz danych deklarując szereg funkcji.

Firma A może jednak obsłużyć tylko niektóre z nich ….->

Język JAVA – adaptery i klasy anonimowe

Adapter

```
interface  InterfejsBazy{
        public boolean polacz(String adres, String nazwaUzytkownika, String haslo);
        public boolean rozlacz();
        public boolean stworzTabele(String []daneAtrybutow);
        public int dodajDaneDoTabeli(String nazwaTabeli, String [][]daneDlaAtrybutow);
        public String wyszukaj(String nazwaTabeli, String warunek);
        public int zmienDane(String nazwaTabeli,String [][]daneDlaAtrybutow);
        public boolean usunTabele(String nazwaTabeli);
        public int szyfrujWymianeDanych(String algorytm);
}//koniec interface InterfejsBazy
public class JediDB implements InterfejsBazy{
        /* Dajemy rzeczywistą implementację wszystkim metodom, ale baza JediDB
        nie będzie obsługiwać szyfrowanego przesyłania danych*/
        // (…) tu implementacja wszystkich metod z wyjątkiem szyfrujWymianeDanych()
        public int szyfrujWymianeDanych(String algorytm){
                return 0; //nic nie robi, tylko zwraca umowny kod wyjścia
        }//koniec szyfrujWymianeDanych()
}//koniec class JediDB
```

A inna baza danych, np. SithDB, może obsługiwać szyfrowanie…

Adapter

A co jeśli będzie baza, która chce być zgodna z naszym standardem, ale obsługiwać będzie tylko 2 metody? Wówczas pozostałe metody trzeba zaimplementować jako puste!

Dla interfejsów, które posiadają więcej niż jedną metodę, programiści często piszą towarzyszącą im klasę (deklarując ją jako abstrakcyjną) implementując wszystkie metody interfejsu jako puste. Taka klasa nazywa się adapterem. Jeśli ktoś pragnie wykorzystać dany interfejs i jednocześnie:

-użyć tylko podzbiór metod zadeklarowanych w interfejsie,

-nie dziedziczy po żadnej klasie,

to może zamiast implementowania interfejsu użyć operacji dziedziczenia po klasie adapteru i nadpisać te puste metody, które pragnie wykorzystać.

Język JAVA – adaptery i klasy anonimowe


```
interface R2D2{
        String wyswietl();    String pokaz();       int policz(int a, int b);
        float generuj();      double srednia();
}//koniec interface R2D2
abstract class R2D2Adapter implements R2D2{ //abstract – tylko do dziedziczenia
        public String wyswietl(){ return ""; }
        public String pokaz(){return"";}
        public int policz(int a, int b){return 0;}
        public float generuj(){return 0.0f;}
        public double srednia(){return 0.0d;}
}//koniec abstract class R2D2Adapter
public class Robot extends R2D2Adapter{
        public String wyswietl(){
                String s = "Tekst ten jest tworem adaptacji R2D2";
                System.out.println(s);          return s;
        }//koniec wyswietl()
        public static void main(String args[]) {
                Robot r = new Robot();          r.wyswietl();
        }//koniec main()
}//koniec public class Robot
```

Kl

```
interface MocGeneracji{        abstract public int moc();  }// koniec interface MocGeneracji
public class Bateria{
        int val;
        Bateria(int poziomEnergii){
                this.val=poziomEnergii;
                System.out.println("Stan baterii= "+this.val);
        }//koniec Bateria()
        public MocGeneracji mGen(){
                return new MocGeneracji() { //klasa implementująca interfejs MocGeneracji
                        public int moc(){
                                return (val / 10);
                        }
                }; //średnik kończy linię instrukcji w ciele metody mGen()
        }//koniec mGen()
        public static void main(String args[]){
                Bateria b = new Bateria(40);   MocGeneracji mg = b.mGen();
                System.out.println("Maksymalna moc generacji= "+mg.moc());
        }//koniec main()
}//koniec public class Bateria
```

Plan wykładu:

1. Podstawy modelowania obiektowego
2. Konstruktory
3. Dziedziczenie, związki pomiędzy klasami, UML
4. Polimorfizm
5. Klasy abstrakcyjne i interfejsy
6. Adaptery, klasy anonimowe
7. Tablice, kolekcje i typ wyliczeniowy

Tablice

Każda tablica w Javie jest obiektem specjalnego typu (typ tablicy). Klasą nadrzędną dla tego typu jest klasa Object.

Jak utworzyć obiekt takiego typu?
Podobnie jak tworzymy obiekt każdej klasy, z użyciem operatora new:
int [] tablica = new int[10]; //tablica 10 elementów typu int
RycerzJedi [] rada = new RycerzJedi[10]; //tablica 10 obiektów

Każdy obiekt tablicy ma może wykorzystać dostępne pola i metody dla typu tablicowego .
Jedno pole:
-length – rozmiar elementów tablicy;
Specjalna metoda (nadpisana z klasy Object):
-clone() – sklonuj obiekt.

Tablice

Co realizuje funkcja clone()?

Metoda ta jest zdefiniowana w klasie Object (jako protected – wykorzystując ją w jakiejś klasie powinniśmy ją nadpisać). Wynikiem jej zastosowania na danym obiekcie powinien być nowy obiekt, który jest kopią (w sensie wszystkich wartości pól) obiektu klonowanego. Oczywiście sklonowany obiekt jest tej samej klasy co obiekt oryginalny

```
public class RycerzJedi implements Cloneable {//interfejs jest tylko znacznikiem typu
        String nazwa;
        String kolor_miecza;
        RycerzJedi(String nazwa, String kolor_miecza){
                this.nazwa=nazwa;
                this.kolor_miecza=kolor_miecza;
        }
        //tu to co we wcześniejszej wersji klasy RycerzJedi
        public Object clone(){
                return new RycerzJedi(this.nazwa, this.kolor_miecza);
        }//koniec clone ()
} // koniec class RycerzJedi
```

d

```
class AtakKlonow {
        public static void main(String[] args) {
                RycerzJedi anakin = new RycerzJedi("Anakin","zielony");
                RycerzJedi vader = (RycerzJedi) anakin.clone(); //moja metoda clone()
                if (anakin!=vader)
                        System.out.println("Vader to klon Anakina, a nie Anakin!");

                int idRycerzy[] = {0, 1, 2};
                int idKlonow[] = idRycerzy.clone(); //metoda clone() typu tablicowego
                if (idRycerzy!=idKlonow)
                        System.out.println("Obiekty tablic nie są takie same po klonowaniu");

                idRycerzy[0]++;
                System.out.println("Wartość pola po zwiększeniu oryginału: "+idKlonow[0]);
                System.out.println("Rozmiar tablicy idRycerzy: "+idRycerzy.length);
        }//koniec main()
}//koniec class AtakKlonow
```

Tablice – klasa Arrays

Przydatny zestaw funkcji dla potrzeb operacji na tablicach zapewnia klasa Arrays w pakiecie java.util. Zdefiniowane funkcje statyczne umożliwiają wykonanie następujących operacji (na różnych typach – polimorfizm):

-Wyszukiwanie elementu w tablicy - binarySearch();
-Sortowanie elementów w tablicy – sort();
-Wypełnianie każdej pozycji tablicy wartością – fill();
-Konwersję tablicy do jej reprezentacji jako jeden ciąg znaków – toString();
-Kopiowanie tablic i podzbiorów tablic – copyOf(), copyOfRange(),
-Porównywanie wszystkich elementów tablic – equals().

d

```
import java.util.Arrays;
public class TabelaJedi{
        public static void main(String []a){
                String [] nazwyJedi=new String[3];
                nazwyJedi[0]="Luke";   nazwyJedi[1]="Anakin";    nazwyJedi[2]="Vader";
                System.out.println("Klasa tablicy to: "+nazwyJedi.getClass().getName());
                //dostępne funkcje w klasie Arrays (s na końcu !!!)
                //szukaj wartości w tablicy
                System.out.println("Wynik: "+Arrays.binarySearch(nazwyJedi,"Anakin"));
                //wyświetl wszystkie wartości tablicy
                System.out.println("Dane w tabeli to: "+Arrays.toString(nazwyJedi));
                //sortuj wartości tablicy
                Arrays.sort(nazwyJedi);
                System.out.println("Dane po sortowaniu: "+Arrays.toString(nazwyJedi));
                //wypełnij tablicę wartościami początkowymi
                Arrays.fill(nazwyJedi,"Unknown");
                System.out.println("Dane w tabeli to: "+Arrays.toString(nazwyJedi));
        }//koniec main()
}//koniec class TabelaJedi
```

Tablice wielowymiarowe

Tablice wielowymiarowe w Javie to tablice tablic. Oznacza to, że najpierw tworzona jest jedna tablica, której elementami są tablice, itd. Załóżmy tablicę dwuwymiarową, która przechowuje obiekty klasy RycerzJedi.

```
RycerzJedi matrycaJedi [][]=new RycerzJedi[2][2];
```

domyślne wartości pól takiej tablicy to null (dla typów podstawowych są to wartości domyślne danego typu, np. 0 dla int).
Aby wypełnić tablicę obiektami musimy do danego miejsca przypisać obiekt, np.:

```
matrycaJedi[0][0]=new RycerzJedi("Luke", "niebieski");
```

Demo ->

d

```
public class MatryceJedi{
        public static void main(String[] a) {
                RycerzJedi matrycaJedi [][]=new RycerzJedi[2][2];
                for (int i = 0; i < matrycaJedi.length; i++)
                        for(int j=0; j<matrycaJedi[0].length;j++)
                                matrycaJedi[i][j]=new RycerzJedi("KlonJedi"+(i+j),"zielony");
                //odczyt ale przy zastosowaniu pętli for each
                for (RycerzJedi[] rZbior : matrycaJedi)
                                for (RycerzJedi r : rZbior)
                                        r.opis();
                //klonujemy obiekt macierzy dwuwymiarowej
                RycerzJedi[][] sith=matrycaJedi.clone();

                //po klonowaniu sith i matrycaJedi to inne obiekty (tylko kopie wartości)
                //ALE te "kopie wartości" to te same tablice (współdzielone)
                if(sith[0]==matrycaJedi[0])
                System.out.println("Tablice pod indeksem 0 to te same obiekty!");
        }// koniec main()
}//koniec class MatryceJedi
```

Tablice– klonowanie tablic wielowymiarowych

Klonowanie tablicy wielowymiarowej – kopia tablicy głównej.

Tablice zawarte w tablicy pozostają te same (referencja!)

```
public class MacierzeKlonow {

  public static void main(String[] args) {
     int idRycerzy[][] = {{0,1,2},{5,6,7}};
     int idKlonow[][] = idRycerzy.clone();
     if (idRycerzy!=idKlonow)
              System.out.println("Obiekty tablic nie są takie same po klonowaniu");

     idKlonow[0][0]++;
     System.out.println("Wartość w tablicy rycerzy po zwiększeniu dla klonu: "+idRycerzy[0][0]);
     System.out.println("Wartość w tablicy klonów po zwiększeniu dla klonu: "+idKlonow[0][0]);
     if (idRycerzy[0]==idKlonow[0]) //te same obiekty tablic?
              System.out.println("Obiekty tablic zawartych w tablicy głównej SĄ takie same!");
  }//koniec main()

}//koniec class MacierzeKlonow
```

Kolekcje obiektów

Jak pokazałem wcześniej tablice w Javie są bardzo przydatne i wygodne w użyciu. Rozmiar tablicy może być ustalony w kodzie źródłowym, jak również jako zmienna (wartość ustalana w czasie wykonywania kodu, np.

```
RycerzJedi [] rycerze;
 int n;
 //czytaj wartość n, np. z klawiatury
rycerze = new RycerzJedi[n]; //etc.
```

Ale co jeśli nie chcemy ustalać rozmiaru tablicy ani na poziomie kodu źródłowego, ani w czasie wykonywania kodu?
Jeśli chcemy mieć "worek" na obiekty, bez podawania i jawnej kontroli jego pojemności (rozmiaru) wówczas możemy zastosować kolekcje obiektów.

Kolekcje obiektów

W modelu wypracowywanym przez ODMG (Object Data Management Group), a później kontynuowanym w ramach prac nad JDO (Java Data Objects) ustalono 3 podstawowe interfejsy, opisujące kolekcje obiektów:

- **List *(lista)*** opisuje listę obiektów (o określonej, dynamicznie tworzonej tablicy);
- **Set *(zbiór)*** opisuje zbiór obiektów niepowtarzalnych.
- **Map *(mapa)*** opisuję kolekcję odwzorowań klucz-wartość. Zarówno klucz jak i wartość są obiektami. Klucze muszą być unikalne.

Każdy interfejs opisuje wymagane zachowanie kolekcji (abstrakcyjne metody), przede wszystkim związane z dodawanie obiektów do kolekcji (***add, put***) i pobieraniem obiektów z kolekcji (***get***). Wykorzystanie określonej kolekcji jest możliwe poprzez zastosowanie klasy implementującej dany interfejs (realizacji metod).

Kolekcje obiektów

Interfejsy List oraz Set rozszerzają interfejs Collection, w którym zadeklarowano podstawowe operacje:

- add() – dodaj obiekt do kolekcji,
- remove(Object o) – usuń obiekt o z kolekcji,
- clear() – usuń wszystkie obiekty z kolekcji,
- size() – pobierz liczbę obiektów w kolekcji,
- toArray() – zamień na macierz, zawierającą wszystkie obiekty,
- iterator() – zwraca obiekt klasy Iterator, umożliwiający przejście po kolejnych obiektach w kolekcji i ich pobranie (metoda next() – zwraca kolejny obiekt kolekcji).

Interfejs Map nie dziedziczy po innym interfejsie. Posiada funkcje clear() i size() oraz inne, w szczególności:

- put(klucz, wartość) – dodaj obiekt "wartość" pod indeksem "klucz",
- get(klucz) – pobierz wartość obiektu, oznaczonego kluczem "klucz".

Klucz jest też obiektem (dowolnej klasy).

Kolekcje obiektów –pakiet java.util

Podstawowe klasy realizujące ideę kolekcji (implementujące interfejsy kolekcji):

- **dla List *(lista):*** Vector i ArrayList (tablica obiektów bez podawania rozmiaru), LinkedList (dodatkowo możliwość operowania na pierwszym i ostatnim elemencie listy, np.: addFirst(), getLast()). Dla realizacji kolekcji typu lista można dodać i pobrać obiekt podając indeks (podobnie jak dla tablic),
- **dla Set *(zbiór):*** HashSet (niepowtarzalne obiekty, funkcja skrótu/hashująca generuje niepowtarzalny kod na bazie tożsamości obiektów {adres, referencja do obiektu}), TreeSet (niepowtarzalne obiekty, a dodatkowo posortowane – konieczność implementacji interfejsu Comparable, czyli zdefiniowania funkcji compareTo()),
- **dla Map *(mapa):*** Hashtable i HashMap (proste odwzorowanie obiekt klucz->obiekt wartość), TreeMap (posortowana mapa według obiektów klucza – podobnie jak dla TreeSet).

d

```
import java.util.*;
/** Zakładamy, że w tym samym pakiecie (katalogu) jest klasa
 *  RycerzJedi z wcześniejszych przykładów.
 */
public class ZbiorJedi{
        public static void main(String [] a){
                RycerzJedi luke=new RycerzJedi("Luke", "niebieski");
                RycerzJedi obi=new RycerzJedi("Obi-wan", "zielony");
                //Podajemy w kolekcji klasę składowanych obiektów – parametr
                Vector<RycerzJedi> lista=new Vector<RycerzJedi>();
                lista.add(luke);  lista.add(obi);   lista.add(luke); //powtarzający się obiekt
                HashSet<RycerzJedi> zbior=new HashSet<RycerzJedi>();
                zbior.add(luke);  zbior.add(obi);  zbior.add(luke); //powtarzający się obiekt
                for(RycerzJedi r:zbior){ //podobnie dla Vector  :lista
                        r.opis();
                }//koniec for
                System.out.println("Rozmiar kolekcji Vector:  "+lista.size());
                System.out.println("Rozmiar kolekcji HashSet: "+zbior.size());
        }//koniec main()
}//koniec class ZbiorJedi
```

Kolekcje obiektów

Jak zapewnić właściwe sortowanie obiektów w zbiorach (TreeSet) oraz mapach (TreeMap)?

1. Klasa obiektów, które są składowane w kolekcji musi implementować interfejs Comparable
2. Implementacja interfejsu wymaga definicji metody compareTo(), która musi zwrócić wartości 0, 1 lub -1, w zależności od wyniku porównania dwóch obiektów (ich pól): równy, mniejszy i większy.

Dla danej klasy trzeba wybrać te pola (pole), które będą porównywane i określać będą unikalność obiektu (taki sam wynik funkcji equals() i compareTo(); czyli true i 0 dla tych samych wartości pól porównywanych obiektów).

Język JAVA – tablice i kolekcje obiektów

d

```
import java.util.*;
class AdeptJedi implements Comparable{
        private int id; //wybieramy ten atrybut jako wyróżnik unikalności - klucz
        AdeptJedi(int id){
                this.id=id;
        }//koniec AdeptJedi()
        public boolean equals(Object o) {
                if (!(o instanceof AdeptJedi))  return false;
                AdeptJedi n = (AdeptJedi)o;
                return (n.id==id)?true:false;
        }//koniec equals()
        public int compareTo(Object o) {
                if (!(o instanceof AdeptJedi))  return -1;
                AdeptJedi n = (AdeptJedi)o;
                int por;
                if(n.id==id)  por=0;  else if(n.id<id)  por=1;  else   por=-1;
                return (por);
        }//koniec compareTo()
        public String toString(){  return ""+id;    }
}//koniec class AdeptJedi
```

d

```
public class UnikalniRycerze{

        public static void main(String []a){
                TreeSet <AdeptJedi>ts=new TreeSet<AdeptJedi>();
                AdeptJedi jacek=new AdeptJedi(2);//wartość 2
                AdeptJedi wieslaw=new AdeptJedi(1);//wartość 1<2
                AdeptJedi ktos=new AdeptJedi(2);//ta sama wartość 2
                ts.add(jacek);
                ts.add(wieslaw);
                ts.add(ktos);
                for(AdeptJedi aj : ts){
                        System.out.println("Rycerz: "+aj);
                }
        }//koniec main()

}//koniec class UnikalniRycerze
```

Wynik: posortowane według ***id*** DWA obiekty. Jeśli mamy obiekty w liście

```
import java.util.*;
class RycerzSith{
        String nazwa;
        RycerzSith(String n){
                nazwa=n;
        }
        public String toString(){       return nazwa;       }
}//koniec class RycerzSith
public class KolekcjaSith{
        public static void main(String []a){
                HashMap<String,RycerzSith> kol=new HashMap<String,RycerzSith>();
                RycerzSith anakin=new RycerzSith("Darth Vader");
                RycerzSith duku=new RycerzSith("Darth Duku");
                kol.put("Anakin",anakin);
                kol.put("Duku", duku);
                System.out.println("Anakin to: "+kol.get("Anakin"));
                System.out.println("Duku to: "+kol.get("Duku"));
        }//koniec main()
}//koniec class KolekcjaSith
```

Typ wyliczeniowy

Typ wyliczeniowy (enumeration) w najprostszej postaci przypomina zbiór stałych o określonych wartościach, przy czym identyfikator i wartość to jedno i to samo oznaczenie.

```
public  enum Typ {JEDI, SITH, ZWYKLY};
//JEDI to identyfikator i wartość
```

Zbiór wartości przetwarzany jest jak kolekcja:

```
for (Typ t : Typ.values())  System.out.println(t);
```

wraz z możliwością ograniczenia zakresu (kolekcje EnumSet, EnumMap):

```
for (Typ t : EnumSet.range(Typ.JEDI, Typ.SITH)) System.out.println(t);
```

Typ wyliczeniowy

Typ wyliczeniowy w Javie to rodzaj klasy, może być zatem definiowany przez pola, metody i konstruktor (prywatny, dostępny wewnątrz)

```
enum Ocena{
        //wywołanie konstruktora z argumentem
        NIEDOSTATECZNA(2), DOSTATECZNA(3), DOBRA(4), BARDZO_DOBRA(5);
        private Ocena(int value) { this.value = value; }
        private final int value; //wartość pola ustawiana tylko jeden raz - final
        public int value() { return value; }
}//koniec enum Ocena

public class TestOcen {
        public static void main(String[] a) {
                for (Ocena o : Ocena.values())
                        System.out.println(o + ": \t"+ o.value());
        }//koniec main()
}//koniec class TestOcen
```

Typ wyliczeniowy

Z elementem wyliczanym (obiektem) można związać metodę…

```
import java.util.*;

public enum WyliczoneOperacje {

        SUMA {int eval(int x, int y) { return x + y; } },
        ROZNICA {int eval(int x, int y) { return x - y; }};

        abstract int eval(int x, int y);

        public static void main(String args[]) {
                int x = 10;
                int y = 5;
                for (WyliczoneOperacje wo : WyliczoneOperacje.values())
                        System.out.println(wo+" "+x+" i "+y+ " = " +        wo.eval(x, y));

        }
}//koniec class/enum WyliczoneOperacje
```

Język JAVA – typ wyliczeniowy ENUM


```
import java.util.*;
enum Ocena{
        NIEDOSTATECZNA(2), DOSTATECZNA(3), DOBRA(4), BARDZO_DOBRA(5);
        private Ocena(int value) { this.value = value; }
        private final int value;
        public int value() { return value; }
}//koniec enum Ocena
public class OcenyJedi {
        public static void main(String[] args) {
          Ocena o;   //wczytaj ocenę z bazy danych, załóżmy, że
          o=Ocena.NIEDOSTATECZNA;
          switch(o) {
            case NIEDOSTATECZNA:
                    System.out.println("Ocena "+o.value()+" oznacza, że NIEZALICZYŁEŚ");
                    break;
            default:
                    System.out.println("ZALICZYŁEŚ");
          }//koniec switch(o)
        }//koniec main()
}//koniec class OcenyJedi
```

Co dalej?

Przedstawiona część wykładu dotycząca modelu obiektowego to wybrane przeze mnie elementy, które uważam za kluczowe. Nie wyczerpuje to jednak w pełni możliwości modelu obiektowego w Javie. Zainteresowanych odsyłam do specyfikacji języka Java oraz dedykowanych podręczników.

Teraz – zapraszam na kolejny wykład nr 5, w którym poznamy możliwości wykorzystania grafiki i budowania interfejsu graficznego w Javie.

Zapraszam na wykład 5