POLITECHNIKA GDAŃSKA

Język Java podstawy

Jacek Rumiński

Język Java podstawy

Jacek Rumiński

Katedra Inżynierii Biomedycznej,
Wydział Elektroniki, Telekomunikacji i Informatyki
Politechnika Gdańska

1. Klasy abstrakcyjne i interfejsy

2. Tablice, macierze, kolekcje

Abstrakcja

Jedno z głównych założeń paradygmatu obiektowego (1. abstrakcja, 2.dziedziczenie, 3. hermetyzacja, 4. polimorfizm).
Abstrakcja oznacza określenie modelu pewnej klasy obiektów (o wspólnych cechach i zachowaniu) **nie podając jednocześnie szczegółów realizacji opisu cech i zachowania, jeśli nie są konieczne**.

Przykładowo Jacek Rumiński, Janina Nowak, Adam Skywalker to 3 konkretne osoby, które mogę zamodelować klasą **Czlowiek**. W klasie tej umieszczę tylko te cechy i funkcje, które są właściwe (wspólne) dla analizowanej grupy obiektów (tj. ludzi).

Jeśli określone zachowanie jest specyficzne dla ludzi ale może być inne np. dla kobiety i mężczyzny wówczas mogę:

- utworzyć metodę modelującą zachowanie w klasie **Czlowiek** (np. unosiKamien()),
- utworzyć klasy **Kobieta** i **Mezczyzna,**
- utworzyć w nich metody o tych samych nazwach i wywołaniu (np. unosiKamien()), które będą przesłaniały (nadpisywały) odziedziczoną metodę o danej nazwie z klasy **Czlowiek**.

Abstrakcja

Jeśli określone zachowanie jest specyficzne dla ludzi ale może być inne np. dla kobiety i mężczyzny wówczas mogę:

- utworzyć metodę modelującą zachowanie w klasie **Czlowiek** (np. unosiKamien()),
- utworzyć klasy **Kobieta** i **Mezczyzna,**
- utworzyć w nich metody o tych samych nazwach i wywołaniu (np. unosiKamien()), które będą przesłaniały (nadpisywały) odziedziczoną metodę o danej nazwie z klasy **Czlowiek**.

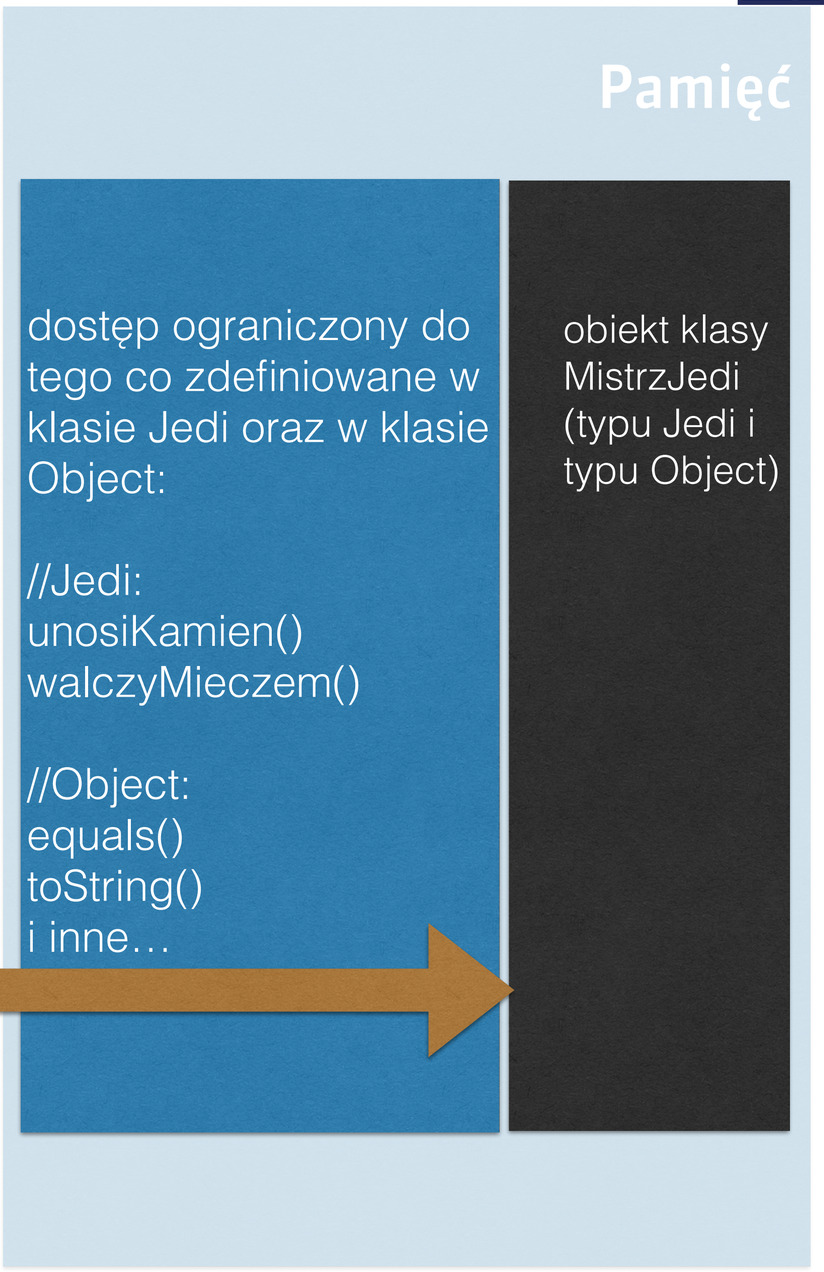
Zauważmy, że w takim postępowaniu:

- metoda unosiKamien() jest zadeklarowana we wszystkich 3 klasach,
- oznacza to, że metoda o tej nazwie (chociaż definicja może być różna) jest WIDOCZNA (zakładamy np. użycie dostępu public) dla obiektów wszystkich 3 klas,

Ponadto zauważmy, że możliwe jest tzw. rzutowanie w górę (ang. upcasting):

```
class Czlowiek {//tu kod m.in. z definicją metody unosiKamien()}
class Kobieta extends Czlowiek{ //tu kod oraz nadpisanie metody unosiKamien()}
class Mezczyzna extends Czlowiek{ //tu kod oraz nadpisanie metody unosiKamien()}
(...)
Czlowiek c1= new Kobieta();
Czlowiek c2= new Mezczyzna();
c1.unosiKamien(); //-> wywołana metoda zdefiniowana w klasie Kobieta
c2.unosiKamien();//-> wywołana metoda zdefiniowana w klasie Mezczyzna
```

```
class Jedi{
        public void unosiKamien(){
                System.out.println("Tracę moc... i kamień spadł");
        }//koniec unosiKamien()
        public void walczyMieczem(){
                System.out.println("Walczę mieczem.");
        }//koniec walczyMieczem(){
}//koniec class Jedi
public class MistrzJedi extends Jedi{
        public void unosiKamien(){
                System.out.println("Unoszę kamień! Czuję moc!");
        }//koniec unosiKamien()
        public void emitujeEnergie(){
                System.out.println("Takie pomarańczowe błyskawice!");
        }//koniec emitujeEnergie()
        public static void main(String []atr){
                Jedi j=new MistrzJedi();
                j.unosiKamien(); //która metoda się wykona?
                j.walczyMieczem();
        }//koniec main()
}//koniec class MistrzJedi
```



Abstrakcja

Abstrakcja w powiązaniu z hermetyzacją związana jest z tym również, że ukrywamy przed otoczeniem rodzaje klas potomnych.

Tworzymy zestaw metod, poprzez które otoczenie komunikuje się (wysyła komunikaty) z naszym obiektem. Zestaw takich metod nazywany jest interfejsem (nie ważne jak działa, ważne jak wywołać i co można uzyskać!). Interfejs zatem to zbiór dostępnych metod jakie można wywołać dla danego obiektu.

Tworząc interfejs czasami nie ma możliwości zdefiniowana działania (zbioru instrukcji) dla pewnej klasy ogólnej. Wówczas używamy:

-klasy częściowo abstrakcyjne (abstract class),
-klasy w pełni abstrakcyjne (interface) .

Klasy abstrakcyjne

Co to jest klasa abstrakcyjna?
➢ skutek: nie można utworzyć obiektu dla klasy abstrakcyjnej,
➢ przyczyna: co najmniej jedna metoda jest abstrakcyjna, lub klasa jest zadeklarowana jako abstract.

Co to znaczy, że metoda jest abstrakcyjna?
Metoda jest abstrakcyjna, jeśli nie posiada swojej realizacji, czyli brak jest jej definicji (instrukcji) nawet definicji pustej (pusty blok kodu {}).

Przykładowo:
```
public int przyspiesz(double silaNacisku);  //średnik bez {} – brak treści
```

Istnieje słowo kluczowe abstract, które umożliwia oznaczenie metody jako abstrakcyjnej.

```
public abstract int przyspiesz(double silaNacisku);
```

```
class Jedi{
		public void unosiKamien(){

		}//koniec unosiKamien()
		public void walczyMieczem(){
			System.out.println("Walczę mieczem.");
		}//koniec walczyMieczem()
}//koniec class Jedi
public class MistrzJedi extends Jedi{
		public void unosiKamien(){
			System.out.println("Unoszę kamień! Czuję moc!");
		}//koniec unosiKamien()
		public void emitujeEnergie(){
			System.out.println("Takie pomarańczowe błyskawice!");
		}//koniec emitujeEnergie()
		public static void main(String []atr){
			Jedi j=new MistrzJedi();
			j.unosiKamien(); //która metoda się wykona?
			j.walczyMieczem();
		}//koniec main()
}//koniec class MistrzJedi
```

Język Java - podstawy
Klasy abstrakcyjne i interfejsy

```
abstract class Jedi{
		public abstract void unosiKamien();  //metoda abstrakcyjna()
		public void walczyMieczem(){
			System.out.println("Walczę mieczem.");
		}//koniec walczyMieczem()
}//koniec class Jedi
public class MistrzJedi extends Jedi{
		public void unosiKamien(){
			System.out.println("Unoszę kamień! Czuję moc!");
		}//koniec unosiKamien()
		public void emitujeEnergie(){
			System.out.println("Takie pomarańczowe błyskawice!");
		}//koniec emitujeEnergie()
		public static void main(String []atr){
			Jedi j=new MistrzJedi();
			j.unosiKamien(); //która metoda się wykona?
			j.walczyMieczem();
		}//koniec main()
}//koniec class MistrzJedi
```

Po co klasy abstrakcyjne?

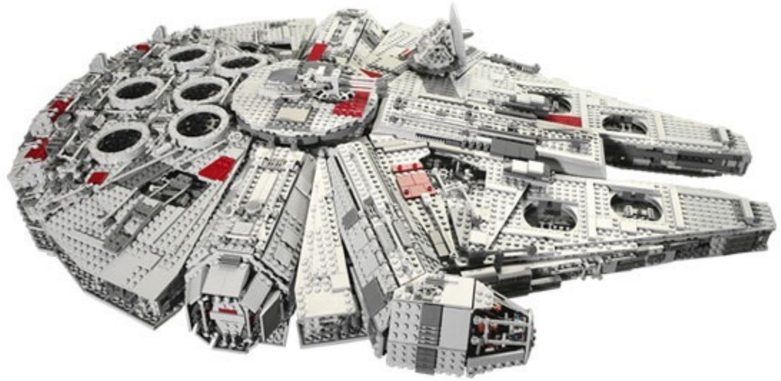
Tworząc model (klasę) ogólny pewnej grupy bytów (obiektów) możemy określić wiele cech i rodzaje zachowania. Niektóre jednak zachowanie jest określane dopiero w bardziej szczegółowym modelu (klasie). Przykładowo zamodelujmy statek:

```
abstract class Statek{
        int numerStatku;
        int liczbaDzial;
        long predkoscMax;
        public abstract int polePowierzchni();//od typu statku zależeć będzie pole
        public void informacje(){
                System.out.println("Liczba dział = "+liczbaDzial);
                System.out.println("Prędkość maksymalna = "+predkoscMax);
                System.out.println("Numer identyfikacyjny = "+numerStatku);
        }

} // koniec abstract class Statek{
```

Po co klasy abstrakcyjne?

Statek (kosmiczny) może być typem okrętu o powierzchni w kształcie trójkąta (Gwiezdny Niszczyciel) czy w kształcie elipsy (Sokół Millenium).



źródło: www.lego.com

ALE UWAGA!!!
Programista będzie mógł również w przyszłości utworzyć nowe klasy (typy okrętów) o rodzaju "Statek". Wówczas będzie musiał dokonać implementacji metody abstrakcyjnej, czyli w procesie dziedziczenia nadpisać metodę abstrakcyjną tworząc jej definicję (czyli zapisując instrukcje obliczenia pola powierzchni danego typu statku).

Implementujemy metody abstrakcyjne.

```
class GwiezdnyNiszczyciel extends Statek{
		int wysTrojkata;
		int dlgPodstawy;
		GwiezdnyNiszczyciel(int numer){
			numerStatku=numer;
		}// koniec GwiezdnyNiszczyciel()
		public int polePowierzchni(){//nadpisanie i implementacja metody abstrakcyjnej
			return (wysTrojkata*dlgPodstawy/2);
		}//koniec polePowierzchni()
}// koniec class GwiezdnyNiszczyciel
class SokolMillenium extends Statek{
		int szer;
		int dlg;
		SokolMillenium(int numer){
			numerStatku=numer;
		}//koniec SokolMillenium()
		public int polePowierzchni(){//nadpisanie i implementacja metody abstrakcyjnej
			return (dlg*szer);
		} //koniec polePowierzchni()
} // koniec class GwiezdnySokol
```

Przykład wykorzystania

```
public class Flota{
    public static void main(String args[]){
        GwiezdnyNiszczyciel gw1 = new GwiezdnyNiszczyciel(1);
        gw1.wysTrojkata=200;
        gw1.dlgPodstawy=500;
        gw1.liczbaDzial=6;
        gw1.predkoscMax=100;
        SokolMillenium gs1 = new SokolMillenium(1);
        gs1.dlg=40;
        gs1.szer=15;
        gs1.liczbaDzial=3;
        gs1.predkoscMax=120;
        Statek s1=(Statek) gw1; Statek s2=(Statek) gs1; //upcasting – zawężanie definicji
        s1.informacje(); //metoda informacje() jest zdefiniowana w klasie Statek
        //metody polePowierzchni() zadeklarowano w klasie Statek
        System.out.println("Pole Niszczyciela to: " + s1.polePowierzchni() +" m(2)");
        s2.informacje();
        System.out.println("Pole Sokoła to: " + s2.polePowierzchni() +" m(2)");
    }
}// koniec public class Flota
```

Interfejsy

Możemy wyobrazić sobie klasę, złożoną z samych metod abstrakcyjnych. Po co?
Jeżeli wyobrazimy sobie szereg funkcji jakie mają realizować obiekty, lecz nie chcemy podawać w jaki sposób wówczas możemy zadeklarować zbiór metod, a inni niech je zaimplementują.

Weźmy na przykład radio. Jedną z funkcji radia jest regulacja siły głosu. Przykładowo: pokrętło w lewo - ciszej, w prawo – głośniej. Taką funkcję mogę przykładowo zapisać jako:

```
void zmienSileGlosu(int skok);// skok<0, ciszej; skok>0, głośniej
```

Jest to metoda abstrakcyjna. Jej realizacja będzie dostarczana przez producenta konkretnego modelu/egzemplarza radia.

Interfejsy

Jeśli inny system (obiekt) pragnie wykorzystać metodę nie musi znać jej realizacji (działania), a jedynie wywołanie, argumenty oraz typ wartości zwracanej.
Przykładowo:
obiekt "Jacek Rumiński" klasy Czlowiek wywołuje metodę zmienSileGlosu() obiektu radio X, przesuwając pokrętło w lewo. Obiekt "Jacek Rumiński" nie musi wiedzieć jaki tok operacji (zmian) będzie wywołany przez metodę (np. nie musi wiedzieć jak zmieni się prąd kolektora, tranzystora numer 123, w układzie ...).

Dlatego właśnie zbiór takich metod tworzy interfejs!

W Javie zaproponowano specjalne słowo kluczowe interface, oznaczające klasę w pełni abstrakcyjną.

Interfejsy

Interfejs można zdefiniować (określić zbiór abstrakcyjnych metod) i zaimplementować (podać definicje wszystkich metod w klasie implementującej interfejs). Jeśli w implementacji nie podamy wszystkich metod (zostanie chociaż jedna bez implementacji) wówczas taka klasa będzie abstrakcyjna. Interfejsy oprócz metod abstrakcyjnych mogą zawierać jedynie stałe.

```
interface MieczJedi {
        static final String TYP="świetlny";
        abstract void dzwiek();
        abstract float moc(int oslabienie);
} // koniec interface MieczJedi
class BronLukea implements MieczJedi{
        public void dzwiek(){   System.out.println("zzzzzzZZZZZZzzzzzzz");  }
        public float moc(int oslabienie){
                float moc_miecza= (mocGeneracji / (dlugosc * r * r * 3.1456f) ) / oslabienie;
                return moc_miecza;
        }
}// koniec class BronLukea
```

Interfejsy

Przypomnijmy: klasa w Javie może dziedziczyć tylko po jednej klasie na raz (A extends B, ale nie ~~C extends D,E~~)

Klasa może implementować wiele interfejsów: A implements B, C

Podsumowując, klasa może dziedziczyć po jednej klasie i implementować wiele interfejsów, np.

A extends B implements C,D

Czyli tworząc model swojej klasy mogę wykorzystać (implementować) WYMAGANIA zdefiniowane w wielu innych interfejsach (modelach).

1. Klasy abstrakcyjne i interfejsy

2. Tablice, macierze, kolekcje

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 (utwórz drugi obiekt o tych samych wartościach danych -> danych tablicy).

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().

Tablice– przykład

```
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");
```

Tablice– przykład

```
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();

        }// koniec main()
}//koniec class MatryceJedi
```

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

Dostępne są podstawowe TRZY RODZAJE kolekcji obiektów, opisywane przez interfejsy:

- 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 (np. klasa ArrayList dla rodzaju List, HashSet dla rodzaju Set, itp. -> więcej w dokumentacji i innych matieriałach).

Kolekcje List i Set – przykład

```
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
                ArrayList<RycerzJedi> lista=new ArrayList<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 ArrayList:  "+lista.size());
                System.out.println("Rozmiar kolekcji HashSet: "+zbior.size());
        }//koniec main()
}//koniec class ZbiorJedi
```

Zapraszamy na kolejne zajęcia