Język JAVA
podstawy programowania

Jacek Rumiński

Wykład 4, część 2

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

Polimorfizm – wielopostaciowość .

Jedno z głównych założeń paradygmatu obiektowego (1. enkapsulacja, 2.dziedziczenie, 3. polomorfizm, 4. …).

Istnieją różne rodzaje polimorfizmu, ale w zasadzie ***własność ta oznacza, że wartości różnych typów (w tym klas) mogą być:***
-Obsługiwane (jako argumenty) przez takie same wywołanie działania (metody),
-Zwracane przez tak samo wywoływane działania (metody).

"Tak samo" oznacza tutaj identyczną nazwę metody lub oznaczenie operatora.

Polimorfizm – wielopostaciowość .

Istnieją różne rodzaje i definicje polimorfizmu. Dla potrzeb naszego wykładu rozpatrzymy trzy postaci polimorfizmu:

- polimorfizm parametryczny,
- polimorfizm ad-hoc,
- polimorfizm przez przesłanianie.

Polimorfizm parametryczny oznacza wykorzystanie parametru zamiast typu danych. Definiujemy klasę, metodę z parametrami, które później mogą być różnie wykorzystane przez podstawienie za parametr klasy lub typu prostego.

Polimorfizm ad-hoc oznacza wykorzystanie skończonej liczby typów danych w różnych konfiguracjach metody o tej samej nazwie, np. suma(int a, int b) oraz suma(double a, double b).

Polimorfizm przez przesłanianie oznacza nadpisywanie metod w procesie dziedziczenia -ta sama metoda wykonywać będzie co innego.

Polimorfizm przez przesłanianie (method overriding)

Polimorfizm przez przesłanianie oznacza nadpisywanie metod w

```
class RycerzykJedi{
        public void walczyMieczem(int moc){
                System.out.println("Walczę z siłą : "+moc);
        } //koniec walczyMieczem()
} //koniec class RycerzykJedi
public class  SuperMistrzJedi extends RycerzykJedi{
        public void walczyMieczem(int moc){//ta sama deklaracja/sygnatura metody
                moc=moc+10; //nadpisujemy treść metody z klasy RycerzykJedi
                System.out.println("Walczę z siłą : "+moc);
        }//koniec walczyMieczem()
        public static void main(String a[]){
                SuperMistrzJedi mj = new SuperMistrzJedi();
                mj.walczyMieczem(20);
        }//koniec main()
}//koniec public class SuperMistrzJedi
```

Kod programu: SuperMistrzJedi.java

```
class RycerzykJedi{
        int wiek=30;
        //eksperyment: dodaj final po public i skompiluj. Co się stanie i dlaczego?
        public void walczyMieczem(int moc){
                System.out.println("Mam "+wiek+" lat.Walczę z siłą : "+moc);
        } //koniec walczyMieczem()
} //koniec class RycerzykJedi

public class  SuperMistrzJedi extends RycerzykJedi{
        int wiek=40;//tylko po co na nowo definiować ten sam identyfikator, lepiej wiek=40
        public void walczyMieczem(int moc){//ta sama deklaracja/sygnatura metody
                moc=moc+10; //nadpisujemy treść metody z klasy RycerzykJedi
                System.out.println("Mam "+wiek+" lat.Walczę z siłą : "+moc);
        }//koniec walczyMieczem()
        public static void main(String a[]){
                SuperMistrzJedi mj = new SuperMistrzJedi();
                mj.walczyMieczem(20);
        }//koniec main()
}//koniec public class SuperMistrzJedi
```

Polimorfizm ad-hoc

Wiele metod (konstruktorów), operatorów o tej samej nazwie lecz z

```
public class  MalyJedi {
        public String sumujMoc(int a, String b){
                return (a+b); //przeciążenie operatora dodawania (dodanie liczby i tekstu)
        }
        public void walczyMieczem(int moc){//argument typu int
                System.out.println("Walczę z siłą : "+moc);
        }//koniec walczyMieczem()
        public void walczyMieczem(String moc){//argument typu String
                System.out.println("Walczę z siłą : "+moc);
        }//koniec walczyMieczem()
        public static void main(String a[]){
                MalyJedi mj = new MalyJedi();
                mj.walczyMieczem(20); mj.walczyMieczem("trzydzieści");
        }//koniec main()
}//koniec public class MalyJedi
```

Polimorfizm ad-hoc

Stosowanie wielu konstruktorów oraz wielu metod o tej samej nazwie jest dobrą praktyką umożliwiającą wykonanie danej operacji bez względu na rodzaj typu danych (metody powinny zrealizować ewentualną konwersję).

Poniżej pokazano fragment dokumentacji klasy String, opisujący przeciążoną funkcję indexOf, zwracającą pozycję wystąpienia znaku lub ciąg znaków w danym łańcuchu znaków, będącym wartością obiektu klasy String.

int	**indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character.
int	**indexOf**(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
int	**indexOf**(String str) Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf**(String str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

Polimorfizm ad-hoc

Poniżej pokazano fragment dokumentacji klasy String, opisujący listę konstruktorów (specjalnych metod o tej samej nazwie).

Constructor Summary

String()
Initializes a newly created String object so that it represents an empty character sequence.

String(byte[] bytes)
Constructs a new String by decoding the specified array of bytes using the platform's default charset.

String(byte[] bytes, Charset charset)
Constructs a new String by decoding the specified array of bytes using the specified charset.

String(byte[] ascii, int hibyte)
Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.*

String(byte[] bytes, int offset, int length)
Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.

String(byte[] bytes, int offset, int length, Charset charset)
Constructs a new String by decoding the specified subarray of bytes using the specified charset.

String(byte[] ascii, int hibyte, int offset, int count)
Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.*

String(byte[] bytes, int offset, int length, String charsetName)
Constructs a new String by decoding the specified subarray of bytes using the specified charset.

String(byte[] bytes, String charsetName)
Constructs a new String by decoding the specified array of bytes using the specified charset.

Polimorfizm parametryczny

Napiszmy konstruktory, metody tak, aby zamiast typów danych były parametry (na wzór zmiennych, których wartość, czyli rodzaj typu, będzie ustalony później).

```
public class SzablonJedi<T>{ //T jest parametrem – zmienną typu
        private T wartosc;
        SzablonJedi (T t){
                wartosc=t;
        }//koniec SzablonJedi()
        T pobierz() {
                return wartosc;
        }//koniec pobierz()
        public static void main(String args[]){
          SzablonJedi<String> szablon = new SzablonJedi<String>("JACEK");//T to String
          System.out.printf("Imie: %s \n",szablon.pobierz());
          SzablonJedi<Integer> szablonik = new SzablonJedi<Integer>(new Integer(33));
          System.out.printf("Wiek: %2d \n",szablonik.pobierz());     //T to Integer
        }//koniec main()
}//koniec class SzablonJedi<T>
```

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

Abstrakcja

Jedno z głównych założeń paradygmatu obiektowego (1. enkapsulacja, 2.dziedziczenie, 3. polomorfizm, 4. abstrakcja).

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. Będę wywoływał określone im funkcje i opis cech (właściwe dla człowieka).

Dopiero jeśli jest konieczność wywołania specyficznego zachowania obiektu Janina Nowak (kobieta!), związanego z tym, że jest to obiekt bardziej szczegółowej klasy (niż Człowiek) wówczas korzystając z dziedziczenia utworzę odpowiednią klasę (Kobieta), funkcję i obiekt.

Abstrakcja

Podstawowa zasada abstrakcyjności oznacza tworzenie klas powiązanych z pojęciami ogólnymi (dla których brak konkretnej realizacji) jak *Czlowiek, Kobieta, RycerzJedi*,… Nie istnieje *RycerzJedi* (to tylko pojęcie ogólne). Może istnieć konkretna (unikalne wystąpienie w czasie i miejscu) osoba, będąca typu *RycerzJedi* (posiada te cechy i zachowanie, które jest wspólne dla danego typu).

Abstrakcja wymaga zatem:

1. Jeśli obiekt jest klasy B, która dziedziczy po klasie A, ale wszystkie niezbędne funkcje zadeklarowane są już w klasie nadrzędnej A, wówczas powinien być przetwarzany (rzutowany) jako obiekt klasy A zamiast B (**upcasting**):

class B extends A; A b = new B();

zawsze używaj klasy nadrzędnej jeśli nie ma potrzeby stosować klasy bardziej szczegółowej

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

Abstrakcja

Abstrakcja związana jest z tym również, że ukrywamy przed otoczeniem nie tylko rodzaje klas potomnych, ale również klas zawartych w danej klasie (kompozycja!). Jeśli klasa *Statek*, składa się z klas obiektów klas *Silnik, Dziala, Kabina*, itd., i nie ma potrzeby ujawniać tej informacji na zewnątrz klasy *Statek*, to stosując zasadę enkapsulacji ukrywamy dostęp do tej informacji.

Jak to zrobić – *tworzymy zestaw metod, poprzez które otoczenie komunikuje się (wysyła komunikaty) z naszym obiektem klasy Statek*!

Zestaw takich metod nazywany jest interfejsem (**nie ważne jak działa, ważne jak wywołać i co można uzyskać!**).

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 dodatkowe oznaczenie metody jako abstrakcyjnej.

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

Po co klasy abstrakcyjne?

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

```
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!!!

Język JAVA – abstrakcja


```
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
```

Język JAVA – abstrakcja


```
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
```

Znaczenie

Upcasting – rzutowanie typu "w górę" w hierarchii dziedziczenia (generalizacji).

A extends B; A a = new B();

*Obiekt "a" jest co najmniej typu **A** (klasy **A**).* Wywołanie konstruktora ***B()*** powoduje utworzenie nowego obiektu klasy ***B*** i jego zapis w pamięci pod pewnym adresem. Dostęp do tego obiektu określony jest przez referencję "a", która jest co najmniej typu ***A*** (jest oczywiście typu ***B***, który jest specjalizacją typu ***A***). Wykonując upcasting ograniczamy dostępny do wywołania zestaw metod obiektu.

ALE: jeżeli metoda w klasie ***A*** jest abstrakcyjna i w klasie ***B*** jest ona zdefiniowana, wówczas obiekt "a" może wywołać taką metodę bo w pamięci jest tak naprawdę obiekt klasy ***B*** ze zdefiniowaną metodą (zadeklarowaną w ***A***). Jeśli metoda nie byłaby zadeklarowana w ***A*** wówczas nie można jej użyć.

Język JAVA – abstrakcja


```
public class Armada{
        public static void main(String args[]){
                GwiezdnyNiszczyciel gw1 = new GwiezdnyNiszczyciel(1);
                SokolMillenium gs1 = new SokolMillenium(1);
                Statek s1=(Statek) gw1;
                Statek s2=(Statek) gs1;
                System.out.println("Klasa s1 to "+s1.getClass().getName());
                System.out.println("Klasa s2 to "+s2.getClass().getName());
                if(s1 instanceof Statek)
                        System.out.println("s1 to Statek");
                if(s2 instanceof Statek)
                        System.out.println("s2 to Statek");
                if(s1 instanceof GwiezdnyNiszczyciel)
                        System.out.println("s1 to GwiezdnyNiszczyciel");
                if(s2 instanceof SokolMillenium)
                        System.out.println("s2 to SokolMillenium");
        }//koniec main()
}// koniec public class Armada
```

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

Tylko po co te interfejsy….
Wyobraźmy sobie następujący problem. Tworzymy system operacyjny JacekOS z obsługa druku na drukarce. Tylko na jakiej?

Interfejsy

Określmy funkcję drukowania:

```
interface Drukarka {
        public void drukuj (String txt);
} // koniec interface Drukarka
```

Załóżmy teraz, że firma Jacek Company, wyprodukowała drukarkę XS23Jet. Producent implementuje usługę druku:

```
public class XS23Jet implements Drukarka {
        public void drukuj (String txt){
                ustawGlowice();
                System.out.println("Drukuje XS23Jet: "+txt);
        }//koniec drukuj()
        private void ustawGlowice(){
                System.out.println("Ustawiam specyficzne parametry X23SJet");
        }//koniec ustawGlowice()
} // koniec interface Drukarka
```

Interfejsy

Załóżmy teraz, że piszemy aplikację dla danego systemu operacyjnego JacekOS, z możliwością druku. Jak wysterować drukarkę nie podając kodu specyficznego dla danej drukarki w programie aplikacji?

```
public class EdytorJedi {
        public static void main(String []a) {
                try{
                        /* wczytaj klasę o nazwie podanej jako pierwszy parametr
                           wywołania programu a[0] ; c – referencja do obiektu
                           reprezentującego wczytaną klasę*/
                        Class c=Class.forName(a[0]);
                        Object o=c.newInstance(); //utwórz obiekt tej nieznanej klasy
                        Drukarka d=(Drukarka) o; //obiekt jest co najmniej typu Drukarka
                        d.drukuj(a[1]); //wydrukuj tekst podany jako drugi parametr…
                }catch (Exception e){
                        System.out.println("Brak klasy sterownika: "+e);
                }//koniec catch()
        }//koniec main()
} // koniec EdytorJedi
```

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