PROGRAMOWANIE W JAVIE

Andrzej Marciniak

Uniwersytet Zielonogórski
Instytut Sterowania i Systemów Informatycznych

Programowanie obiektowe w Javie:

- Klasy
- Dziedziczenie
- Polimorfizm
- Pakiety i interfejsy
- Podstawowe klasy Javy

Deklaracje klas - modyfikatory właściwości

abstract oznacza że nie można tworzyć obiektów danej klasy (jest ona abstrakcyjna)

final oznacza że nie można z niej wyprowadzać podklas

public oznacza że można wykorzystać klasę w dowolnym miejscu (domyślnie klasa jest dostępna tylko wewnątrz pakietu w którym została zdefiniowana)

Modyfikatory właściwości pól danych

static oznacza pole danych klasy, a nie obiektu

final definiuje stałą, którą można przypisać tylko przy inicjalizacji obiektu

transient pola tego typu nie są trwałą częścią obiektu i nie zostają zachowane przy archiwizacji obiektu. Stosowane są do zaznaczania pól nie podlegających szeregowaniu (*object serialization*) czyli zapisywaniu i odczytywaniu obiektów bajt po bajcie w komunikacji szeregowej

volatile oznacza, że pole może być modyfikowane asynchronicznie, przez konkurencyjne wątki w programach wielowątkowych(zostanie omówiony na wykładzie poświęconym programowaniu współbieżnemu)

Modyfikatory właściwości metod

static oznacza metodę klasy, która może operować tylko na zmiennych statycznych

final definiuje metodę, której nie można przedefiniować w podklasie

abstract metoda nie posiadająca implementacji i będąca członkiem klasy abstrakcyjnej,

native oznacza metodę zaimplementowaną w innym języku niż Java i dołączoną za pomocą Java Native Interface,

synchronized oznacza metodą będącą sekcją krytyczną (segment kodu w programie, w którym następuje dostęp do tej samej danej z różnych wątków-zostanie omówiony na wykładzie poświęconym programowaniu współbieżnemu)

Dziedziczenie

Relację dziedziczenia między nadklasą i podklasą wyrażamy za pomocą frazy ze słowem extends .

```
class Punkt{int x; int y;
            Punkt(){x=1;y=1;} //niezbędny! - omówię dalej
            Punkt(int x, int y){this.x=x; this.y=y;}
           }

class Punkt3D extends Punkt {
    int z;
    Punkt3D(int x, int y, int z)
    {   this.x=x;   //inicjuje zmienną odziedziczoną
        this.y=y;   //inicjuje zmienną odziedziczoną
        this.z=z;
    }
    Punkt3D()
    {
        this(1,1,1);
    }
}
```

Słowo kluczowe super

Do pól i metod nadklasy odwołujemy się za pomocą słowa kluczowego super. W podklasie, słowo kluczowe super reprezentuje wtedy nazwę nadklasy. Dzięki temu możemy odwoływać się do przykrytych składników nadklasy (pól danych i metod na nowo zdefiniowanych w podklasie) przy dziedziczeniu.

```
class Punkt{int x; int y;
            Punkt(int x, int y){this.x=x; this.y=y;}
           }
class Punkt3D extends Punkt {
    int z;
    Punkt3D(int x, int y, int z)
    {   super(x,y);  //wywołanie konstruktora Punkt(X,Y)
        this.z=z;
    }
    Punkt3D()
    {
        this(1,1,1);
    }
}
```

Zasady dziedziczenia

- pola danych i metody z modyfikatorem private nie są dziedziczone,
- pola danych i metody nadklasy bez modyfikatora dostępu mogą być dziedziczone tylko przez podklasy zdefiniowane w tym samym pakiecie,
- podklasy "nie dziedziczą" tych pól danych i metod nadklasy, które mają taką samą nazwę jak pola i metody zadeklarowane w podklasie. Pola danych podklasy ukrywają pola danych nadklasy, a metody podklasy przesłaniają metody nadklasy. Dostęp do składowych nadklasy jest za pomocą słowa super.
- gdy w konstruktorze podklasy nie ma wywołania konstruktora nadklasy, Java domyślnie przyjmuje wywołanie super() — konstruktor bezparametrowy nadklasy, jeśli taki konstruktor nie istnieje, to sygnalizowany jest błąd.

Przykrywanie metod - przykład

```
class Punkt{int x; int y;
            //... konstruktory ...
     public void Zeruj()
        {X=0;Y=0; System.out.println("Zerowanie Punktu");
        }
}
class Punkt3D extends Punkt {
    int z;
    //...konstruktory...
    public void Zeruj()
    {super.Zeruj(); z=0; System.out.println("Zerowanie Punktu3D");
}
// ----------wywołanie metod----------------------------
Punkt a=new Punkt(); Punkt3D b=new Punkt3D();
a=b; //można przypisać - odwrotne przypisanie byłoby błędne
a.Zeruj(); //wywołuje metodę dla klasy 3D
```

Przykrywanie metod - wielokrotne dziedziczenie

Odwołanie typu super.super.NazwaMetody() przy wielokrotnym dziedz. nie jest poprawne(ale można przeprowadzić konwersję referencji this).

```
class Raz {String m_SNazwa= "Raz"; //zmienna
    String s()  {  return "1"; } //funkcja
}
class Dwa extends Raz { String m_SNazwa= "Dwa";
    String s() { return "2"; }
}
class Trzy extends Dwa { String m_SNazwa= "Trzy";
    String s() { return "3"; }
    void test()
    {System.out.println("s()=\t\t\t"+s());
     System.out.println("m_SNazwa=\t\t"+m_SNazwa+"\n...");
     System.out.println("super.s()=\t\t"+super.s());
     System.out.println("super.m_SNazwa=\t\t"+super.m_SNazwa+"\n....");
     System.out.println("((Dwa)this).s()=\t"+((Dwa)this).s());
     System.out.println("((Dwa)this).m_SNazwa=\t"+((Dwa)this).m_SNazwa);
     System.out.println(".........");
     System.out.println("((Raz)this).s()=\t"+((Raz)this).s());
     System.out.println("((Raz)this).m_SNazwa=\t"+((Raz)this).m_SNazwa);
```

Przykrywanie metod - przykład

Wywołanie

```
.....Trzy trzy = new Trzy();
     trzy.test();
```

spowoduje wyświetlenie:

s()= 3

m_SNazwa= Trzy

...

super.s()= 2

super.m_SNazwa= Dwa

....

((Dwa)this).s()= 3

((Dwa)this).m_SNazwa= Dwa

.....

((Raz)this).s()= 3

((Raz)this).m_SNazwa= Raz

Przykrywanie metod- wielokrotne dziedziczenie

Dla pól danych użycie słowa kluczowego super lub konwersji do klasy Raz lub Dwa powoduje wypisanie na ekranie wartości pola m_SNazwa z odpowiedniej klasy. Natomiast dla metod, konwersja typu ((Dwa)this).s() jest równoważna wywołaniu this.s().

Dzieje się tak dlatego, że w Javie każda metoda, która nie jest statyczna (static) lub prywatna (private) jest wirtualna (ang. virtual). Dlatego wywołanie metody s() klasy Raz: (((Raz)this).s()) jest przekształcane w wywołanie przedefiniowującej ją metody s() z klasy Trzy. Tę właściwość nazywa się dynamicznym rozdzielaniem metod .

Dynamiczne rozdzielanie metod

(ang. dynamic method dispatch) umożliwia uzyskiwanie polimorfizmu czasu przebiegu - wywoływanie metod należących do podklas przez metody klas zdefiniowanych w istniejących już i skompilowanych bibliotekach, bez konieczności ponownego ich kompilowania i z zachowaniem tego samego interfejsu.

Podczas działania programu interpreter Javy ustala klasę obiektu, na jaki wskazuje zmienna. Jeżeli nie wskazuje ona na obiekt klasy za pomocą którego została zadeklarowana, lecz na jej podklasę, a jednocześnie podklasa ta zawiera wywoływaną metodę, to ona właśnie jest wykonywana, a nie przykryta metoda z nadklasy.

Dynamiczne rozdzielanie metod - przykład

```
class A
{   void napis()
        {
            System.out.println("Klasa A");
        }
}
class B extends A
{   void napis()
        {
            System.out.println("Klasa B");
        }
}
class rozdzial
{
    public static void main(String args[])
    {A a = new B();
     a.napis();     //wyświetli 'Klasa B'
    }
}
```

Słowo kluczowe final

- definiuje klasy, które nie mogą być dziedziczone,
- definiuje metody, których nie wolno przykrywać w podklasach,
- umożliwia deklarowanie stałych (wykład 2)

Zmiennym finalnym nie jest przydzielana pamięć osobno w każdym obiekcie (podobnie jak dla statycznych). Metody finalne są często wykonywane szybciej niż zwykłe gdyż kompilator może wstawić ich kod bezpośrednio do kodu metody wywołującej, tworząc kod wewnętrzny (*inline*).
Kompilator Javy kopiuje tylko niewielkie metody (opłacalne). Nigdy nie powstaje kod wewnętrzny metod innych niż finalne, gdyż nie byłoby możliwe uzyskanie polimorfizmu czasu przebiegu za pomocą dyn. rozdzielania metod.

Klasy finalne - bezpieczeństwo

Popularnym mechanizmem wykorzystywanym przez hackerów do łamania systemów jest tworzenie własnych podklas klas oryginalnych, a następnie podmienianie ich. Podklasa taka "wygląda" jak oryginalna, ale może powodować zupełnie inne działanie, powodując straty lub ułatwiając hackerowi dostęp do poufnych informacji. Klasy finalne uniemożliwiają to!!!

Przykład. Klasa String w pakiecie java.lang, która jest na tyle ważna dla operacji kompilatora i interpretera, że należy zapewnić że gdy kiedykolwiek metoda lub obiekt używa klasy String to jest to zawsze java.lang.String. To gwarantuje że wszystkie łańcuchy nie mają niespójnych, niepożądanych czy nieprzewidywalnych właściwości.

Klasy i metody abstrakcyjne

Niekiedy definiujemy klasę reprezentującą jakąś abstrakcyjną koncepcję, opisującą pewne własności wspólne dla reprezentowanej abstrakcji. Przykładem takiej koncepcji abstrakcyjnej niech będzie pojęcie: "figura". Tworzenie obiektu klasy figura nie ma sensu, ze względu na jego abstrakcyjność, ale już podklasy figury, np. kwadrat, trójkąt czy koło mogą być instancjowane.

Klasa abstrakcyjna może zawierać metody abstrakcyjne, które nie zawierają implementacji. W ten sposób klasa abstrakcyjna definiuje interfejs programistyczny, który musi znaleźć się w nie-abstrakcyjnych jej podklasach.

Klasy abstrakcyjne - przykład

```
abstract class GraphicObject {
    int x, y;
    . . .
    void moveTo(int newX, int newY) {   //metoda zwykła
        x=newX;y=newY;
    }
    abstract void draw();   //bez implementacji - metoda abstrakcyjna
}
class Circle extends GraphicObject {
    void draw() {
        . . .       //wymaga implementacji
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        . . .       //wymaga implementacji
    }
}
```

Klasy i metody abstrakcyjne-właściwości

- Nie jest wymagane, aby klasa abstrakcyjna zawierała metody abstrakcyjne. Jednakże każda klasa, która ma metodę abstrakcyjną, lub która nie implementuje metod abstrakcyjnych dziedziczonych z nadklasy, musi być zadeklarowana jako klas abstrakcyjna.
- Nie wolno deklarować abstrakcyjnych konstruktorów oraz abstrakcyjnych metod statycznych, choć klasa abstrakcyjna może zawierać ich nie-abstrakcyjne wersje.
- Każda podklasa klasy abstrakcyjnej musi implementować wszystkie metody abstrakcyjne nadklasy, chyba że sama została zadeklarowana jako abstrakcyjna.

Interfejsy

Interfejs jest kolekcją publicznych metod abstrakcyjnych (bez implementacji). Różni się on od klasy abstrakcyjnej (KA) następująco:

- nie może implementować żadnych metod (KA może),
- nie jest częścią hierarchii klas (KA jest) - nie powiązane klasy mogą mieć takie same interfejsy,
- klasa może mieć wiele interfejsów, ale tylko jedną nadklasę - interfejs może dziedziczyć z innych interfejsów, ale nie może dziedziczyć z klas,
- jeśli istnieją w interfejsie pola danych, to są one domyślnie publiczne, finalne i statyczne (KA-nie tylko takie),
- klasa, która implementuje interfejs, musi posiadać definicje wszystkich metod zadeklarowanych w interfejsie (KA -tylko abstrakcyjnych), w przeciwnym wypadku klasa taka będzie klasą abstrakcyjną.

Definiowanie interfejsu

[modyfikator] interface NazwaInterfejsu [extends listaInterfejsów]
{ *członkowie klasy nie mogą być: volatile, synchronized, transient,private, protected*
. . .
}

```
interface Kolekcja
{ int MAXIMUM = 200;
  void dodaj(Object obj);
  Object znajdz(Object obj);
  int liczbaObiektow();
}
```

Implementowanie interfejsu

```
class Wektor implements Kolekcja
{   private Object obiekty[] = new Object[MAXIMUM];
    private short m_sLicznik = 0;
    public void dodaj(Object obj)
    {
       obiekty[m_sLicznik++]=obj;
    }
    public Object znajdz(Object obj)
    {
        for (int i = 0; i<m_sLicznik;i++)
        {  if (obiekty[i].getClass() == obj.getClass())
            { System.out.println("Znaleziono obiekt   klasy "
                                              + obj.getClass() );
              return obiekty[i];
            }
        }
        return null;
    }
    public int liczbaObiektow()
    {      return m_sLicznik;
    }
}
```

Interfejs jako typ

```
class Test
{
    public void funkcja_testowa(Kolekcja kolekcja, int delta)
    {
        //... ciało funkcji
    }
public static void main(String args[])
    {
        Kolekcja zmienna = new Wektor();
        zmienna.liczbaObiektow(); // dynamiczne wywołanie
    }
}
```

Metody są wybierane dynamicznie w czasie przebiegu. Ponieważ odszukiwanie metod w czasie działania programu wpływa negatywnie na szybkość jego działania, to w miejscach, w których ma ona decydujące znaczenie, należy unikać deklarowania zmiennych jako odwołań do interfejsów.

Rozrastanie interfejsów

Ze względu na funkcjonalność tworzonych bibliotek, interfejsy nie powinny się rozrastać! Dodanie nowych metod do interfejsu $Kolekcja$ spowodowałoby konieczność przedefiniowania wszystkich klas po nim dziedziczących (inaczej byłyby abstrakcyjne). Inni programiści przestaliby używać takich bibliotek - zamiast tego lepiej zadeklarować podinterfejs.

```
public interface NowaKolekcja extends Kolekcja {
    void currentValue(String tickerSymbol, double newValue);
}
```

Pakiety

Pakiety w Javie są pewnym zbiorami klas i interfejsów dostarczającymi ochronę dostępu dla swoich składników oraz zapewniającymi zarządzanie nazwami (brak konfliktów nazw).
Przykład: dane są klasy i interfejsy zadeklarowane w różnych plikach

```
//klasa zadeklarowana w pliku  Graphic.java
public abstract class Graphic {
    . . .
}
//klasa zadeklarowana w pliku Circle.java
public class Circle extends Graphic implements Draggable {
    . . .
}
//klasa zadeklarowana w pliku Draggable.java
public interface Draggable {
    . . .
}
```

Pakiety - przykład

Aby utworzyć pakiet, należy umieścić wyrażenie package na początku każdego pliku źródłowego który ma być powiązany w pakiecie, np.

```
//plik  Graphic.java
package graphics;

public abstract class Graphic {
    . . .
}
```

Uwaga! Jeśli plik *.java* zawiera więcej niż jedną klasę, tylko jedna z nich może być publiczna i jej nazwa musi być taka sama jak nazwa pliku. Jeśli plik źródłowy nie zawiera słowa package , wszystkie jego klasy i interfejsy znajdują się w pakiecie default package , będącym pakietem dla małych i tymczasowych aplikacji. Po skompilowaniu, dla każdej klasy tworzone są pliki z kodem bajtowym o nazwach: NazwaKlasy.class

Tworzenie pakietów

Struktura pakietów musi być skorelowana ze strukturą katalogów na dysku. Na przykład pliki zawierające instrukcję:

```
package java.awt.image;
```

powinny znajdować się w katalogu $java \setminus awt \setminus image$. Katalog główny struktury katalogów odpowiadających poszczególnym pakietom określa się za pomocą zmiennej środowiskowej CLASSPATH (jako wartość można podać większą liczbę katalogów).Jeśli zmienna ta przyjmuje wartość: CLASSPATH $= C{:}\setminus Windows \setminus java \setminus classes;$ to ostatecznie pliki przykładu znajdą się w katalogu: $C{:}\setminus Windows \setminus java \setminus awt \setminus image$.
Uwaga: Niektóre firmy umieszczają biblioteki w Internecie, stosując następującą konwencję: $com.nazwafirmy.nazwapakietu$.

Import pakietów

Import pojedynczego elementu pakietu odbywa się za pomocą instrukcji import , np:

```
import java.awt.image.mojaklasa;
```

a wszystkich elementów pakietu za pomocą gwiazdki (asterisk):

```
import java.awt.image.*;
```

Import pakietów-uwagi

- umieszczenie gwiazdki w większej liczbie instrukcji import powoduje wydłużenie czasu kompilowania programu, szczególnie gdy importowane pakiety zawierają wiele klas - nie ma to jednak wpływu na szybkość działania skompilowanego programu,
- deklaracja importu nie oznacza włączania do pliku tekstu zawartego w pliku źródłowym pakietu(nie jest odpowiednikiem dyrektywy include preprocesora z C++)
- ważna jest kolejność deklaracji:najpierw deklaracja pakietu, po niej deklaracje importu, po czym definicje klas.
- standardowe klasy Javy należące do pakietu java.lang są importowane automatycznie (domyślne import java.lang.*;)

Podstawowe klasy Javy - Character

Klasa której instancja przechowuje pojedynczy znak (zapakowuje wartość prymitywnego typu wbudowanego *char* w obiekt).

```
public class CharacterDemo {
    public static void main(String args[]) {
        Character a = new Character('a');
        Character a2 = new Character('a');
        Character b = new Character('b');
        int difference = a.compareTo(b);   //funkcja porównująca
        if (difference == 0) {
        System.out.println("a is equal to b.");
        }
        System.out.println("a is "
            + ((a.equals(a2)) ? "equal" : "not equal")+ " to a2.");
        System.out.println("The character " + a.toString() + " is "
            + (Character.isUpperCase(a.charValue()) ? "upper" : "lower")
            + "case.");
    }
}
```

Wybrane metody klasy Character

Character(char) jedyny konstruktor klasy

int compareTo(char) porównuje numerycznie dwa znaki

boolean equals(Object) porównuje znak do obiektu w parametrze i sprawdza czy są równe,

String toString() konwertuje do Stringu,

char charValue() zwraca wartość obiektu,

static boolean isUpperCase(char) sprawdza czy duża litera,

Podstawowe klasy Javy - String i StringBuffer

Platforma Javy dostarcza dwie klasy obsługujące napisy: *String* i *StringBuffer*. Klasa String jest wykorzystywana do przechowywania stałych napisów (lepsza optymalizacja kodu), a StringBuffer dla modyfikowalnych napisów.

```
public class StringsDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        StringBuffer dest = new StringBuffer(len);

        for (int i = (len - 1); i >= 0; i--) {
            dest.append(palindrome.charAt(i));
        }
        System.out.println(dest.toString());
    }
}
```

Konstruktory klas String i StringBuffer

String(), StringBuffer() inicjalizuje pusty łańcuch,

String(byte []bytes) dekoduje tablicę byte-ów i tworzy łańcuch

String(byte[] bytes, int offset, int length, String charsetName) jw. ale z zastosowaniem tablicy kodowej

String(char[] value) kopiuje elementy z tablicy znaków do łańcucha

String(char[] value, int n, int m) jw. ale tylko od n-tego elementu i m znaków

String(String), StringBuffer(String) kopiujący

String(StringBuffer) jw. ale ze StringBuffer

StringBuffer(int n) konstruuje (allokuje) łańcuch o n elementach

Konstrukcja obiektów typu String i StringBuffer

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
StringBuffer dest = new StringBuffer(len);

String s=new String();

char helloArray []= { 'h', 'e', 'l', 'l', 'o' };
String helloString [] = new String(helloArray);

byte ascii []= { 65,66,67 };
String ascii2=new String(ascii,0);
```

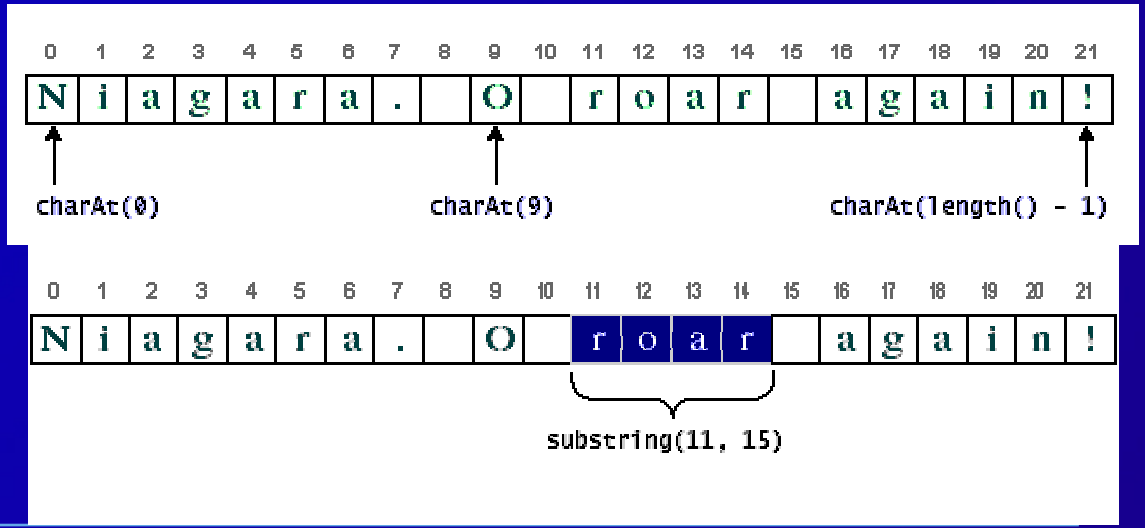
Konkatenacja łańcuchów

```
int age=15;
String s = "On ma " + age + "lat"; // tak naprawdę wykona się:
/* String  s = new StringBuffer("On ma ")
             .append(age)
             .append("lat")
             .toString();*/
```

Jest to spowodowane tym że egzemplarze klasy String są niezmienialne.
Przeciążenie operatora + dla klasy String jest odstępstwem od reguł Javy.

Wydobywanie znaków z łańcuchów

```
String anotherPalindrome = "Niagara. O roar again!";
char aChar = anotherPalindrome.charAt(9);
```



Inne metody klasy String

Pełen zestaw metod klas String i StringBuffer można znaleźć na stronie

`http://java.sun.com/j2se/1.4.1/docs/api/java/lang/String.html`

.

String toUpperCase() konwertuje do dużych liter

char[] toCharArray() konwertuje do tablicy znaków

String trim() zwraca łańcuch bez pustych znaków

String concat(String) konkatenuje łańcuch

int compareTo(String) porównuje leksykalnie łańcuchy (dobre dla sortowania)