

WIEDZA KTÓRA ZMIENIA

Programowanie w języku Java

Inżynieria Oprogramowania
Altkom Akademia

Cele kursu

- Nabycie praktycznych umiejętności programowania aplikacji w języku Java
- Poznanie fundamentów języka Java niezbędnych do dalszego poznawania zaawansowanych technologii
- Pomoc w przygotowywaniu się do certyfikatu programisty (OCPJP)

Zawartość kursu

1. Wprowadzenie do technologii Java

- Platforma Javy
- Narzędzia programistyczne

2. Proste obiekty i typy danych

- Typy proste i obiektowe
- Klasy, metody, argumenty metod

3. Zasady tworzenia aplikacji

- Środowisko programistyczne
- Struktura projektu

Zawartość kursu

4. Podstawy składni języka

- Konstrukcje sterujące, operacje na zmiennych, tablice

5. Koncepcja programowania obiektowego

- Modelowanie obiektowe
- Klasy, interfejsy

6. Obsługa błędów i wyjątków

- Zasady obsługi wyjątków; własne wyjątki

Zawartość kursu

7. Środowisko systemowe i operacje wejścia-wyjścia

- Kolekcje danych
- Klasy systemowe
- Strumienie i operacje plikowe

8. Najczęściej popełniane błędy w Javie

- Sytuacje, na które trzeba zwracać szczególną uwagę

Zawartość kursu

9. Zaawansowane kolekcje oraz wejście-wyjście

- Algorytmy i sortowanie kolekcji
- Dekorowanie strumieni
- Filtr zip
- Klasa Scanner
- Formatowanie liczb i dat

10. Biblioteka standardowa Java SE

- Logowanie
- Wyrażenia regularne
- Kilkuwierszowe przykłady

11. Internacjonalizacja w języku Java i18n

- Tworzenie aplikacji wielojęzycznych

Zawartość kursu

12. Programowanie wielowątkowe

- Cykl życia wątków
- Synchronizacja wątków i ochrona danych

13. Interfejs graficzny

- Podstawowe pojęcia
- Delegacyjny model zdarzeń
- Podstawowe komponenty graficzne

14. Komunikacja z bazami danych JDBC API

- Uzyskiwanie połączenia, odczyt oraz modyfikacja danych

Rozkład kursu

Moduł	Dzień				
	1	2	3	4	5
1. Wprowadzenie do technologii Java	X				
2. Proste obiekty i typy danych	X				
3. Zasady tworzenia aplikacji w Javie	X				
4. Podstawy składni języka		X			
5. Koncepcje programowania obiektowego		X			
6. Obsługa błędów i wyjątków			X		
7. Środowisko systemowe i operacje wejścia-wyjścia			X		
8. Najczęściej popełniane błędy w Javie			X		
9. Zaawansowane kolekcje oraz wejście-wyjście				X	
10. Biblioteka standardowa Java SE				X	
11. Internacjonalizacja w języku Java i18n				X	
12. Programowanie wielowątkowe					X
13. Interfejs graficzny					X
14. Komunikacja z bazami danych JDBC API					X

Przygotowanie uczestników

- Umiejętność programowania w innych językach
 - C/C++
 - Pascal, Delphi
 - Visual Basic
- Umiejętność tworzenia i edycji plików tekstowych

Przedstawienie uczestników

- Imię i nazwisko, firma
- Rodzaj wykonywanej pracy
- Doświadczenie w programowaniu
- Powody uczestnictwa w kursie (oczekiwania)

Plan szkolenia

1 Wprowadzenie do technologii Java

2 Proste obiekty i typy danych

3 Zasady budowania aplikacji w Javie

4 Podstawy składni języka

5 Koncepcje programowania obiektowego

6 Obsługa błędów i wyjątków

7 Środowisko systemowe i operacje we/wy

8 Najczęściej popełniane błędy w Javie

Plan modułu

1

Wprowadzenie do technologii Java

- podstawowe pojęcia
- założenia projektowe języka Java
- wirtualna maszyna Javy
- prosta aplikacja
- dokumentacja kodu i korzystanie z niej

Początki języka

- Rozwijany w Sun Microsystems jako język do programowania urządzeń elektroniki użytkowej
- Premiera w 1995 roku jako język tworzenia aplikacji dla Internetu
- Szybka popularność dzięki dostępności w przeglądarce Netscape
- Pierwsza technologia "ożywiająca" strony Web

Dystrybucje Javy

- Java ME – Java Platform, Micro Edition
 - do tworzenia aplikacji na urządzenia przenośne
- **Java SE – Java Platform, Standard Edition**
 - podstawowa dystrybucja języka
 - umożliwia tworzenie aplikacji z interfejsem graficznym
- Java EE – Java Platform, Enterprise Edition
 - zawiera rozszerzenia do tworzenia aplikacji webowych oraz integracji z innymi systemami

Środowisko pracy

- Dystrybucja Java SE oraz dokumentacja API
 - dostępna za darmo na wiele platform pod adresem
<http://java.sun.com>
- Zawiera narzędzia uruchamiane z linii komend do komplikowania i uruchamiania aplikacji
- Dowolny edytor tekstowy: *Notepad*

Założenia projektowe języka

- Ma być prosty w użyciu, pozbawiony pułapek, w które najczęściej wpadają programiści
- Ma być przenośny między różnymi środowiskami uruchomieniowymi
- Ma gwarantować bezpieczeństwo użytkownikowi, który pobrał aplikacje z sieci

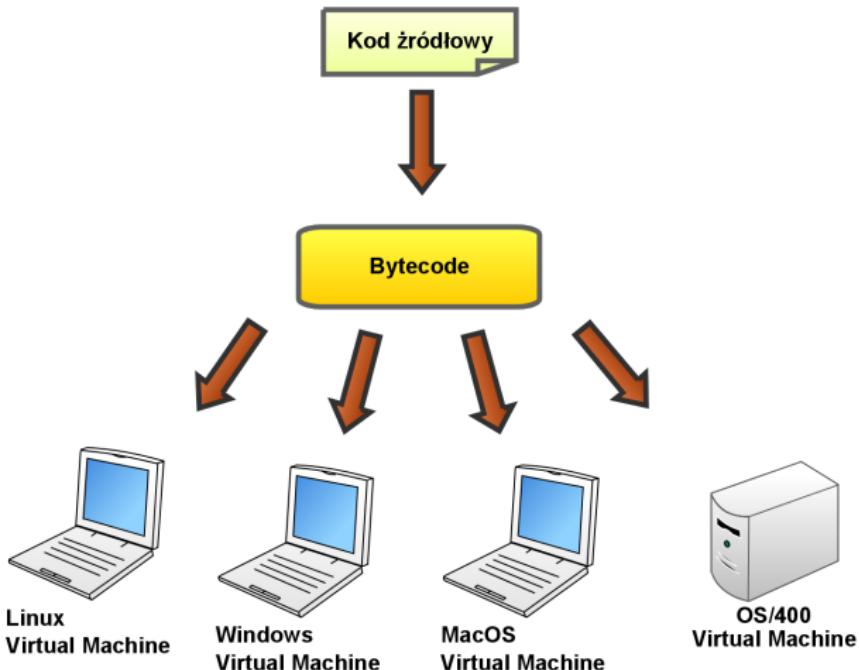
Prosty w użyciu

- Język obiektowy
- Składnia zbliżona do składni C
- Kontrola zakresu i typu zmiennych
- Automatyczne zarządzanie pamięcią
- Dynamiczne ładowanie klas
- Ujednolicony mechanizm dokumentowania

Zarządzanie pamięcią

- Automatyczne zwalnianie pamięci po nieużywanych obiektach
- Mechanizm odzyskiwania nieużytków (*Garbage Collection*) śledzi wszystkie odniesienia (referencje) do obiektów
- Obiekty stają się nieużytkami, jeśli z poziomu aplikacji zostały do nich “zgubione” referencje
- GC działa asynchronicznie — programista nie może bezpośrednio sterować odzyskiwaniem pamięci, a jedynie zasugerować zwolnienie nieużywanych obszarów poprzez wywołanie funkcji `System.gc()`

Przenośność kodu



Koncept JVM

Przenośność kodu

- Raz napisana aplikacja może być uruchamiana na dowolnej platformie sprzętowej posiadającej JVM
- Kod jest przenośny zarówno na poziomie źródeł, jak i kodu skompilowanego
- Aplikacja jest dostosowana do wyglądu graficznego i sposobu obsługi, jaki obowiązuje w danym środowisku uruchomieniowym

Bezpieczeństwo aplikacji

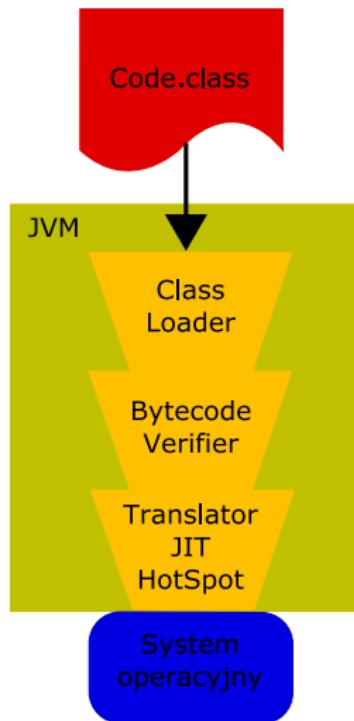
- Podczas komplikacji kodu następuje sprawdzenie składni i kontrola typów
- W trakcie uruchamiania następuje bieżąca kontrola poprawności stosowania zmiennych
- Podczas ładowania klas do *JVM* stosowane są dodatkowe mechanizmy weryfikacji kodu
- Możliwość stosowania *SecurityManagera*
 - Dodatkowy mechanizm umożliwiający kontrolę wykonywanych przez aplikację instrukcji

Java Virtual Machine

- Definicja hipotetycznego komputera
- Definiuje składnię pliku klas, zestaw instrukcji, zestaw rejestrów, stos, układ pamięci
- Implementacja *JVM* realizowana jest jako emulacja software'owa lub sprzętowa
- *JVM* ładuje i wykonuje kod binarny aplikacji napisanej w języku Java

Ładowanie kodu

- Kod aplikacji ładowany jest do JVM poprzez *Class Loader*
- Załadowany kod jest weryfikowany w *Bytecode Verifier*
- Zweryfikowany kod jest tłumaczony na kod maszynowy danej platformy operacyjnej przez *Translator*



Class Loader

- Mechanizm ładowania z sieci lub dysku binarnego kodu klasy do JVM
- Ładowane klasy umieszczane są w odpowiednich przestrzeniach nazwowych
- Możliwość zastosowania reguł bezpieczeństwa dla przestrzeni nazwowych
- Rozdzielenie przestrzeni nazwowych uniemożliwia “podsywanie się” klas

Bytecode Verifier

- Załadowany kod klasy poddawany jest weryfikacji:
 - zgodności ze specyfikacją
 - poprawności stosowanych typów zmiennych
 - poprawności stosowanych konwersji danych
 - poprawności wywoływania metod
 - zachowania integralności systemu

Translator kodu

- Zweryfikowany kod jest poddawany tłumaczeniu na kod maszynowy danego środowiska uruchomieniowego
- Efektywność tłumaczenia kodu poprawiono dzięki:
 - *JIT* — kompilacji w locie — przetłumaczony kod jest zapamiętywany w pamięci
 - *HotSpot* — kod jest optymalizowany dynamicznie pod kątem sposobu jego używania

Pierwsza aplikacja

- Kod źródłowy zapisywany jest w plikach tekstowych
- Nazwa pliku musi być zgodna z nazwą klasy publicznej z rozszerzeniem `.java`
 - Java rozróżnia duże i małe litery
- Pojedynczy plik może zawierać co najwyżej jedną klasę publiczną

Pierwsza aplikacja – najprostszy program

- Plik *HelloWorld.java*

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello");  
    }  
}
```

Pierwsza aplikacja – alternatywa

- Plik *Test.java*

```
public class Test {  
    public static void main(String args[]) {  
        Hello hello = new Hello();  
        hello.sayHello();  
    }  
}
```

- Plik *Hello.java*

```
public class Hello {  
    public void sayHello() {  
        System.out.println("Hello");  
    }  
}
```

Kompilacja

- Zapisany plik źródłowy poddaje się komplikacji
- Zasady komplikacji:
 - jeśli brak źródeł, używane są binaria
 - jeśli brak binariów, używane są źródła
 - jeśli są źródła i binaria, używane są binaria, chyba że źródła są bardziej aktualne

```
javac Test.java
```

Uruchomienie

- Skompilowane klasy przyjmują rozszerzenie `.class`
- Uruchomienie polega na wywołaniu *JVM* wraz ze wskazaniem skompilowanej klasy zawierającej metodę *main*

```
java Test
```

Błędy kodu

- Część błędów kodu wykrywanych jest na etapie kompilacji, np.:
 - błędy składniowe lub dostępność kodu
 - błędy związane ze stosowaniem typów zmiennych, konwersją danych
- Błędy wynikające z logiki działania aplikacji wykrywane są podczas uruchamiania, np.:
 - korzystanie z niezainicjowanych zmiennych
 - przekroczenie indeksów tablicy

Dokumentacja kodu

- Dokumentacja kodu w postaci stron HTML jest automatycznie generowana
- Źródłem informacji jest kod i komentarze
 - definicje klas i składników klasy
 - komentarze dokumentujące `/* * -- -- -- */`
- Komentarz dokumentujący dostarcza dodatkowego opisu elementu przed którym jest umieszczony
- Komentarz dokumentujący może zawierać dodatkowe znaczniki informacyjne

Przykład Javadoc

```
/**  
 * The class <code>Exception</code> and its  
 * subclasses are a form of <code>Throwable</code>  
 * that indicates conditions that a reasonable  
 * application might want to catch.  
 *  
 * @author Frank Yellin  
 * @version 1.32, 11/17/05  
 * @see java.lang.Error  
 * @since JDK1.0  
 */
```

Dokumentacja API Javy

The screenshot shows a Windows Internet Explorer window displaying the Java Platform SE 6 API documentation. The title bar reads "Overview (Java Platform SE 6) - Windows Internet Explorer". The address bar shows the URL "http://java.sun.com/javase/6/docs/api/". The page content is the "Java™ Platform, Standard Edition 6 API Specification". The left sidebar lists "All Classes" and "Packages" under "java.awt". The main content area includes a navigation bar with links for Overview, Package, Class, Use, Tree, Deprecated, Index, Help, PREV, NEXT, FRAMES, and NO FRAMES. A right sidebar displays the "Java™ Platform Standard Ed. 6" logo. The "Packages" section table lists the following packages:

Packages	Description
java.awt	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt.color	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.datatransfer	Provides classes for color spaces.
java.awt.dnd	Provides interfaces and classes for transferring data between and within applications.
java.awt.event	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.font	Provides interfaces and classes for dealing with different types of events fired by AWT components.

Podsumowanie

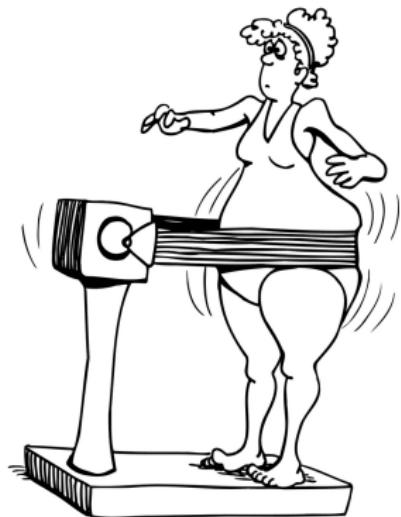
- Czym jest platforma Java
- Główne cechy języka
 - wygodny dla programistów
 - przenośność
 - bezpieczeństwo
- Pierwsza aplikacja
- Korzystanie z dokumentacji

Pytania sprawdzające

- Jakie znasz dystrybucje Javy?
- Jaka jest różnica pomiędzy JRE a JDK?
- Jakie cechy języka Java decydują o prostocie użycia?
- Jaką rolę pełni mechanizm *Garbage Collection*?
- Na jakim poziomie zagwarantowana jest przenośność kodu?
- Jaką rolę pełni wirtualna maszyna Javy (*JVM*)?
- Jakie warunki musi spełniać kod programu zawarty w pliku z rozszerzeniem *.java*?
- Jakich poleceń należy użyć, aby skompilować i uruchomić program w Javie?
- W jaki sposób można utworzyć dokumentację projektu w Javie?

Ćwiczenia

- Ćwiczenie 1.1:
kompilacja i uruchomienie przykładowego programu
- Ćwiczenie 1.2:
*korzystanie z dokumentacji;
modyfikacja, komplikacja i uruchomienie aplikacji*



Plan szkolenia

- 1 Wprowadzenie do technologii Java
- 2 Proste obiekty i typy danych**
- 3 Zasady budowania aplikacji w Javie
- 4 Podstawy składni języka
- 5 Koncepcje programowania obiektowego
- 6 Obsługa błędów i wyjątków
- 7 Środowisko systemowe i operacje we/wy
- 8 Najczęściej popełniane błędy w Javie

Plan modułu

2

Proste obiekty i typy danych

- typy proste
- typy obiektowe
- klasy i ich składniki
- konstruktor klasy
- organizacja klas w pakiety, importy
- kod źródłowy
- argumenty metod
- zakres zmiennych

Typy proste

- Są to **jedynie nieobiektowe** składniki w języku Java
- Z identyfikatorem zmiennej skojarzona jest bezpośrednio jej wartość
- Typy proste służą do reprezentowania:
 - wartości logicznych
 - wartości znakowych
 - wartości całkowitych
 - wartości zmiennoprzecinkowych

Typ logiczny

- Nazwa typu: ***boolean***
- Przechowuje wartość logiczną
- Zakres wartości: ***true, false***
- Przykład deklaracji:

```
boolean prawda = true;  
boolean lightOn = false;
```

Typ znakowy

- Nazwa typu: **char**
- Przechowuje kod znaku w Unicode
- Zakres wartości: 16 bit [0 do 2¹⁶-1]
- Przykład deklaracji:

```
char c = 'c';
char ogonki = '\u0105';
```

Typy całkowite

- Nazwa typu: **byte, short, int, long**
- Przechowuję liczby całkowite ze znakiem
- Zakres wartości:

typ	długość	zakres
<i>byte</i>	8 bit	$-2^7 \div 2^7 - 1$
<i>short</i>	16 bit	$-2^{15} \div 2^{15} - 1$
<i>int</i>	32 bit	$-2^{31} \div 2^{31} - 1$
<i>long</i>	64 bit	$-2^{63} \div 2^{63} - 1$

Typy całkowite – zastosowania

- Przykłady deklaracji (z inicjalizacją):

```
byte b = 15;
int i = -1234;
long len = 12345L;
int bin = 0b101;           // od JSE 7
int octal = 077;
int hex = 0xFF;

long phone = 510_123_456L; // od JSE 7
```

Typy całkowite – zastosowania

- Domyślne literały całkowite są typu *int*
- Operacje arytmetyczne na liczbach całkowitych dokonywane są na zmiennych typu *int*

Typy zmiennoprzecinkowe

- Nazwa typu: ***float*, *double***
- Przechowuję wartości dziesiętne
- Zakres wartości:
 - *float* – 32 bit [IEEE754]
 - *double* – 64 bit [IEEE754]

Typy zmiennoprzecinkowe

- Domyślne literały dziesiętne są typu *double*
- Przykłady deklaracji (z inicjalizacją):

```
float f = 10.1F;  
double big = 6.41e37;  
  
double payment = 12_345.6; // od JSE 7
```

Typ referencyjny *String*

- Nazwa typu: ***String***
- Przechowuje łańcuchy znaków (*char*)
- Jest to typ referencyjny, ale z uproszczoną składnią tworzenia obiektów i łączenia łańcuchów
- Przykład deklaracji:

```
String tekst = "To jest tekst";
```

Typ referencyjny *String*

- Jeśli jednym z argumentów operatora + jest wartość typu *String*, to następne są automatycznie zamieniane na *String*

```
int liczbaLat = 6;  
String tekst = "Ala ma " + liczbaLat + " lat";
```

- Co będzie wynikiem działania: $1+2+"3"+4+5$?

Typ wyliczeniowy *enum*

- Nazwa typu: ***enum***
- Reprezentuje typ wyliczeniowy
- Istnieje możliwość dodawania metod oraz pól. Typ ten rozszerza klasę *Enum* i implementuje wszystkie jego metody
- Przykład deklaracji:

```
enum Season {WINTER, SPRING, SUMMER, FALL}
```

- Możliwość wykorzystania w instrukcji *switch*

Konwersje typów

- Podczas operacji na typach nie może dojść do utraty precyzji
- Sytuacje mogące doprowadzić do utraty precyzji (informacji) muszą być świadomie wskazane przez programistę

Konwersje typów

- Przykłady błędnych operacji:

```
int a = 10L;
/* wartosc 64-bitowa jest zamieniana na 32-bitowa */

float f = 10.0;
/* domyslny literal typu double przypisywany do float -
 * utrata precyzji */

int a = false;
/* nie istnieje bezposrednia konwersja typu boolean na
 * typ calkowity */
```

Konwersje typów

- Domyślna konwersja typów może być przeprowadzona w przypadku pełnego zachowania informacji (precyzji)

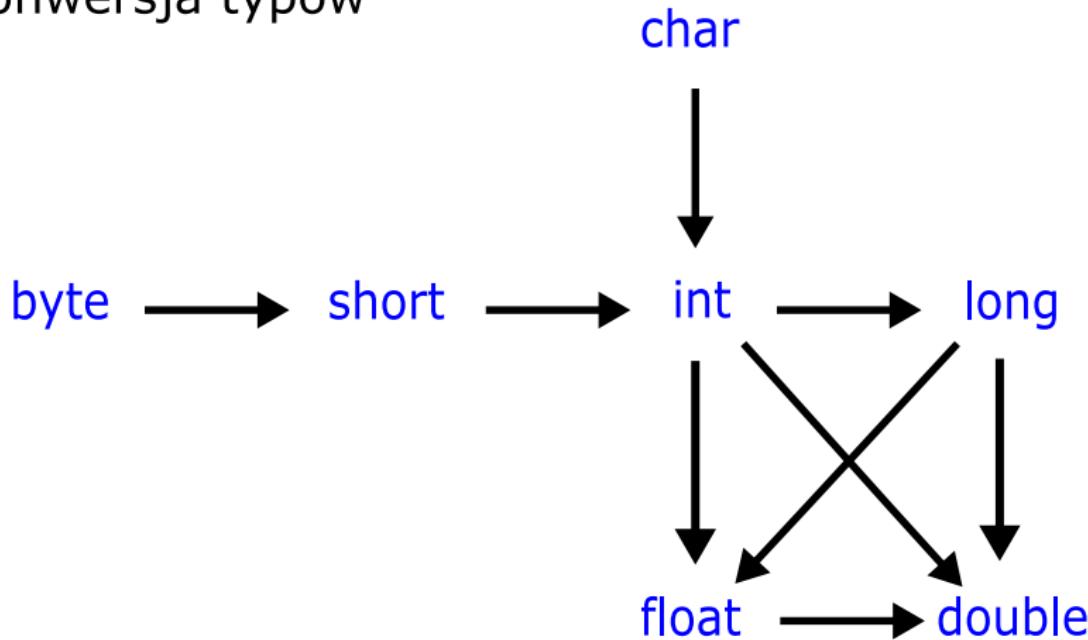
```
long big = 10;  
int val = 'a';  
float dec = 100;
```

- Aby umożliwić operację konwersji typów tam, gdzie zachodzi groźba utraty informacji należy zastosować **rzutowanie**

```
int a = (int) 10L;  
float f = (float) 10.0;
```

Diagram konwersji typów

Konwersja typów



Typy referencyjne

- Typ referencyjny pod nazwą zmiennej – identyfikatorem – kryje odniesienie do obiektu
- Przechowuje “referencję” do obiektu, a nie “zawartość” obiektu
- Jeden obiekt może być zatem dostępny poprzez kilka zmiennych referencyjnych
- Odniesienia do obiektów są śledzone przez mechanizm *Garbage Collection*

Typy referencyjne

- Deklaracja zmiennej referencyjnej składa się z nazwy typu (klasy) oraz identyfikatora:

```
String tekst;  
Date date;
```

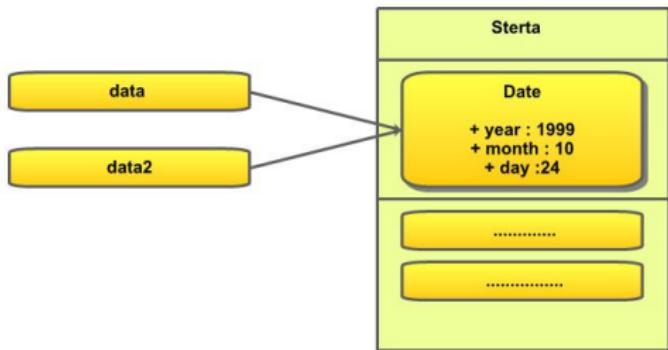
- Zadeklarowana zmienna jest w stanie przechować referencję do obiektu danego typu
- Obiekt określonego typu tworzy się poprzez operator *new*

```
new Date(1999, 10, 24);
```

Typy referencyjne

- Operator `new` alokuje (rezerwuje) pamięć na stercie do przechowywania danych obiektu
- Po utworzeniu obiektu operator `new` zwraca referencję do obiektu, którą można przypisać do zmiennej referencyjnej:

```
Date data = new Date(1999, 10, 24);  
Date data2 = data;
```



Typy referencyjne

- Definicja typu to definicja klasy:

```
public class Date {  
    int year, month, day;  
}
```

- Obiekt powołany na podstawie typu posiada wszystkie składniki klasy
- Dostęp do składników klasy poprzez operator “.”

```
Date data = new Date();  
data.year = 1999;
```

Tworzenie klasy

- Klasa stanowi szablon, z którego tworzone są obiekty
- Definicja klasy zawiera:
 - modyfikator klasy: *public*, <pusty>
 - składniki klasy:
 - atrybuty
 - metody
 - konstruktory
- Składnia:

```
<modyfikator> class <nazwa> {  
    [składniki*]  
}
```

Przykład klasy

```
import java.util.Date;

public class Person {

    // atrybuty
    String name = "Jan";
    Date birth = new Date();
    String pesel;

    // metody
    public void showPesel() {
        System.out.println(pesel);
    }
    public String getInfo() {
        return name + " " + birth + " " + pesel;
    }

    // konstruktory
    public Person(String name, Date birth, String pesel) {
        this.name = name;
        this.birth = birth;
        this.pesel = pesel;
    }
    public Person() { }
}
```

Atrybuty klasy

- Atrybuty stanowią zestaw danych na jakich operuje obiekt
- Atrybutami mogą być zarówno typy proste, jak i typy referencyjne
- Atrybutom mogą być nadane wartości początkowe
- Inne nazwy dla atrybutów: pola, zmienne klasy
- Składnia:

```
<modyfikator> <typ> <nazwa> [= <wartosc>];
```

Atrybuty klasy – przykład

```
// atrybuty  
  
String name = "Jan";  
Date birth = new Date();  
String pesel;
```

Metody

- Metody są opercjami, jakie może wykonywać obiekt na swoich danych
- Metody służą również do ustawiania lub odczytywania wartości atrybutów klasy
- Składnia:

```
<modyfikator> <typ zwracany> <nazwa> ([lista argumentow]) {  
    [operacje*]  
}
```

Metody klasy – przykład

```
// metody

public void showPesel() {
    System.out.println(pesel);
}

public String getInfo() {
    return name + " " + birth + " " + pesel;
}
```

Konstruktor

- Konstruktor jest składnikiem klasy, który jest wykonywany w trakcie tworzenia obiektu
- Może służyć do wykonania operacji przygotowujących obiekt do pracy
- Składnia:

```
<modyfikator> <nazwa klasy> ([lista argumentow]) {  
    [operacje*]  
}
```

Konstruktory klasy – przykład

```
// konstruktory

public Person(String name, Date birth, String pesel) {
    this.name = name;
    this.birth = birth;
    this.pesel = pesel;
}

public Person() { }
```

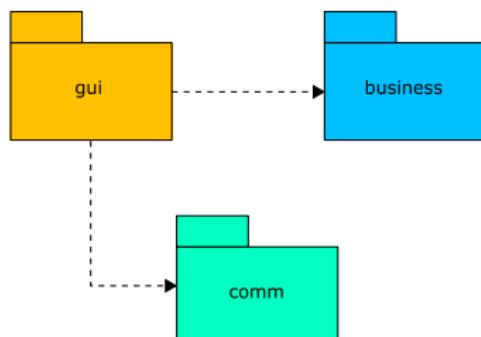
Konstruktor domyślny

- Każda klasa ma konstruktor
- Jeśli konstruktor nie jest zdefiniowany w sposób jawny, w klasie istnieje bezargumentowy konstruktor domyślny
- Konstruktor domyślny nie posiada argumentów
- Zdefiniowanie własnego konstruktora usuwa konstruktor domyślny

```
public Person() { }
```

Pakiety

- Każda klasa może stać się elementem pakietu
- Pakiet ma na celu grupowanie klas
 - o podobnym charakterze funkcjonalnym
 - współpracujących ze sobą nad realizacją danego zadania
 - stanowiących samodzielny moduł systemu
- Pakiet może zawierać w sobie inne pakiety



Pakiety

- Klasa może przynależeć tylko do jednego pakietu
- Klasa może ograniczać dostęp do siebie i swoich składników do poziomu pakietu – *modyfikator pusty*
- Przynależność do pakietu jest zapisywana jako pierwsza znacząca linia w kodzie źródłowym
- Składnia:

```
package <nazwa>[.<nazwa podpakietu>]*;
```

- Przykład:

```
package aplikacja.gui;
```

Import

- Dowolna klasa może korzystać z klas (typów) pochodzących z innych pakietów
- Klasy z innych pakietów należy wskazywać pełną nazwą:

<nazwa_pakietu>.<nazwa_klasy>

np.:

application.gui.FrameManager

- W celu uproszczenia nazewnictwa klas z innych pakietów można zastosować instrukcję *import*

Import

- Import rozszerza przestrzeń nazw danej klasy o wskazany pakiet lub klasę, np.:

```
/* rozszerzenie przestrzeni nazw na cały pakiet */  
import java.util.*;
```

```
/* włączenie wskazanej klasy do przestrzeni nazw */  
import java.text.NumberFormat;
```

-
- Import pakietu nie rozszerza przestrzeni nazw na podpakiety
 - W przypadku konfliktu nazw, klasy muszą być adresowane pełną nazwą

Kod źródłowy

- Plik źródłowy może zawierać:
 - deklarację przynależności do pakietu
 - instrukcje *import* rozszerzające przestrzeń nazw
 - co najwyżej jedną definicję klasy publicznej
 - dowolnie dużo definicji klas pakietowych
- Plik źródłowy nosi nazwę klasy publicznej i rozszerzenie .java
- Składnia:

```
[ package <nazwa> ]  
[ import <pakiet/klasa> ] *  
<definicja_klasy>+
```

Struktura pliku źródłowego

```
package pl.altkom.jpr.mod02;
import java.util.Date;

public class Person {

    // atrybuty
    String name = "Jan";
    Date birth = new Date();
    String pesel;

    // metody, konstruktory ...
}
```

Kompilacja kodu

- Kod uporządkowany w pakiety jest umieszczany w strukturze katalogów odpowiadającej strukturze pakietów
- Komenda kompilatora wskazująca strukturę katalogów

```
javac -d <dir> <nazwa_pliku>
```

- Nazwa klasy zawierającej metodę *main* musi występować w pełnym brzmieniu jako argument JVM

```
java application.gui.FrameManager
```

Argumenty metod

- Metody klasy mogą zawierać listę argumentów
- Składnia listy:

```
<typ> <nazwa> [, <typ> <nazwa>] *
```

- Przykład:

```
public int setDate(int year, int month, int day)
```

Argumenty metod

- Metody są wołane na rzecz istniejącego obiektu:

```
data.tomorrow();  
data.setDay(10);
```

- Wywołanie metody z argumentami powoduje przekazanie do ciała metody **kopii wartości argumentów**

Przekazywanie argumentów do metody

- Zmienna podana jako argument metody jest przekazywana przez jej wartość
- Do metody trafia jej kopia
- Operacje wykonywane na kopii nie mają wpływu na oryginał
- Uwaga:
 - zmienna referencyjna i jej kopią wskazują na **ten sam** obiekt
 - zmiany wykonane na obiekcie poprzez kopię referencji są również widoczne przez oryginał referencji

Przykład

```
public void changeVal(int val) { val = 10; }
public void changeVal(String val) { val = "pies"; }
public void changeVal(Date val) { val.day = 10; }
```

Jaka będzie wartość zmiennych po następujących wywołaniach?

```
int a = 5;
changeVal(a); // a?
```

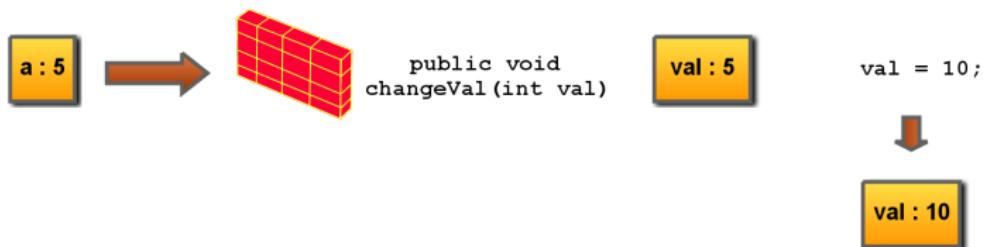
```
String msg = "kot";
changeVal(msg); // msg?
```

```
Date date = new Date(1999, 10, 24);
changeVal(date); // date?
```

Przekazywanie argumentów

```
public void changeVal(int val) { val = 10; }
```

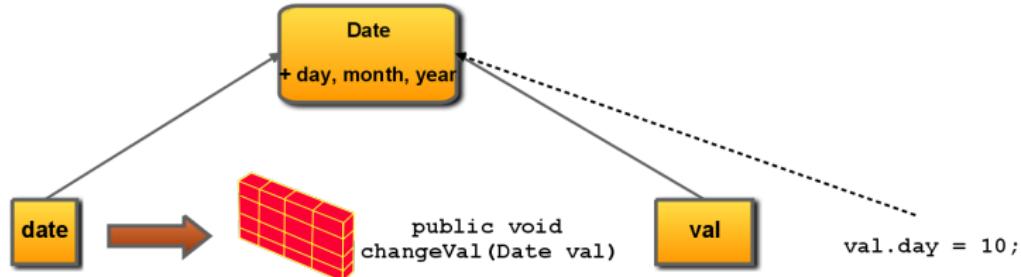
```
int a = 5; changeVal(a);  
System.out.println(a); // ?
```



Przekazywanie argumentów

```
public void changeVal(Date val) { val.day = 10; }
```

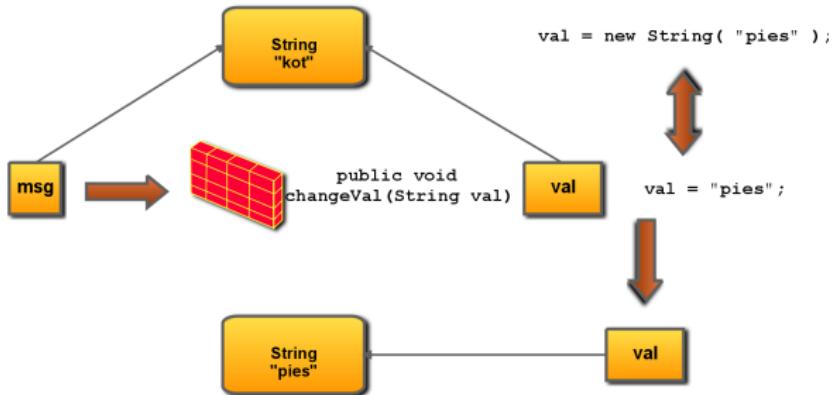
```
Date date = new Date(1999, 10, 24); changeVal(date);  
System.out.println(date.day); // ?
```



Przekazywanie argumentów

```
public void changeVal(String val) { val = "pies"; }
```

```
String msg = "kot"; changeVal(msg);  
System.out.println(msg); // ?
```



Zakres zmiennych

• Zmienne klasy (atrybuty)

- widoczne są dla wszystkich metod i konstruktorów klasy
- ich czas życia jest tożsamy z czasem życia obiektu
- podczas inicjacji obiektu ich wartości są wstępnie ustawiane na wartość domyślną

typ	wartość domyślna
<i>boolean</i>	<i>false</i>
<i>byte, short, int, long, char, float, double</i>	0
typy referencyjne	<i>null</i>

Zakres zmiennych

- Zmienne metod mają zakres lokalny
 - są widoczne tylko wewnątrz metody
 - czas życia jest równoważny czasowi realizacji zestawu instrukcji danej metody
- Każdy blok może mieć swoje zmienne lokalne dostępne wewnątrz bloku
- Atrybuty klasy przesłonięte przez zmienne lokalne mogą być dostępne przez referencję **this**
 - **this** – zawiera referencję do bieżącego obiektu

Zakres zmiennych

```
public class Variables {  
    private int var = 10;  
    public Variables (int var) {  
        this.var = var;  
    }  
    public void method() {  
        int localVar = 10;  
        // ...  
        {  
            int blockVar = 10;  
            localVar = blockVar;  
        }  
        // blockVar = 10; /* blad - zmienna niedostepna */  
    }  
}
```

Podsumowanie

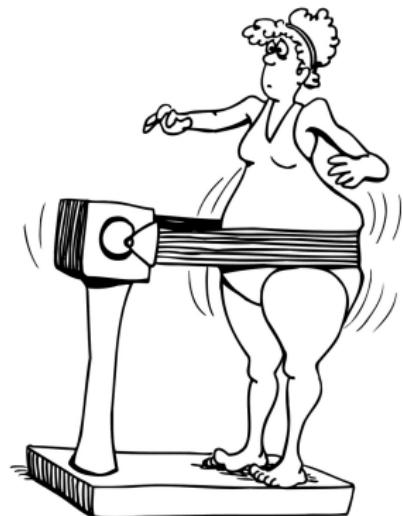
- Typy proste
 - *boolean, char, byte, short, int, long, float, double*
 - konwersja danych
- Typy obiektowe
 - czym jest referencja, typ *enum*
- Składnia klasy i kodu źródłowego:
 - atrybuty, konstruktor i metody klasy
 - pakiety, instrukcja *import*
- Zasięg zmiennych

Pytania sprawdzające

- Wymień wszystkie nieobiektowe składniki w Javie
- Jakiego typu są domyślne literały całkowite i zmiennoprzecinkowe?
- W jakim celu stosujemy rzutowanie?
- Jaka jest różnica pomiędzy zmienną typu prostego, a zmienną typu referencyjnego?
- W jaki sposób tworzymy obiekty?
- Wymień składniki klasy
- Jaka jest składnia konstruktora?
- Jakie cechy posiada konstruktor domyślny?
- Do czego służą pakiety i instrukcje importu?
- Jaka jest poprawna struktura zawartości pliku kodu źródłowego Javy?
- W jaki sposób przekazywane są argumenty do metod?
- W jaki sposób można odwołać się do przesłoniętego atrybutu klasy z poziomu konstruktora lub metody?

Ćwiczenia

- Ćwiczenie 2.1:
przykład przekazywania argumentów do metod – wyjaśnij zachowanie (plik TestVar.java)
- Ćwiczenie 2.2:
typy referencyjne
- Ćwiczenie 2.3:
klasy i metody



Plan szkolenia

- 1 Wprowadzenie do technologii Java
- 2 Proste obiekty i typy danych
- 3 Zasady budowania aplikacji w Javie**
- 4 Podstawy składni języka
- 5 Koncepcje programowania obiektowego
- 6 Obsługa błędów i wyjątków
- 7 Środowisko systemowe i operacje we/wy
- 8 Najczęściej popełniane błędy w Javie

Plan modułu

3

Zasady budowania aplikacji w Javie

- środowisko pracy
- Eclipse IDE
- struktura projektu
- organizacja pakietów oraz klasa główna

Środowisko pracy

- Obecnie na rynku dostępnych jest wiele środowisk programistycznych
- Przykłady
 - darmowe: *Eclipse*, *NetBeans*, *JDeveloper*, *JCreator*
 - płatne: *IntelliJ*, *WebSphere*, *WebLogic Workshop*, *MyEclipse*, *JBuilder* – często posiadają bezpłatne, okrojone wersje

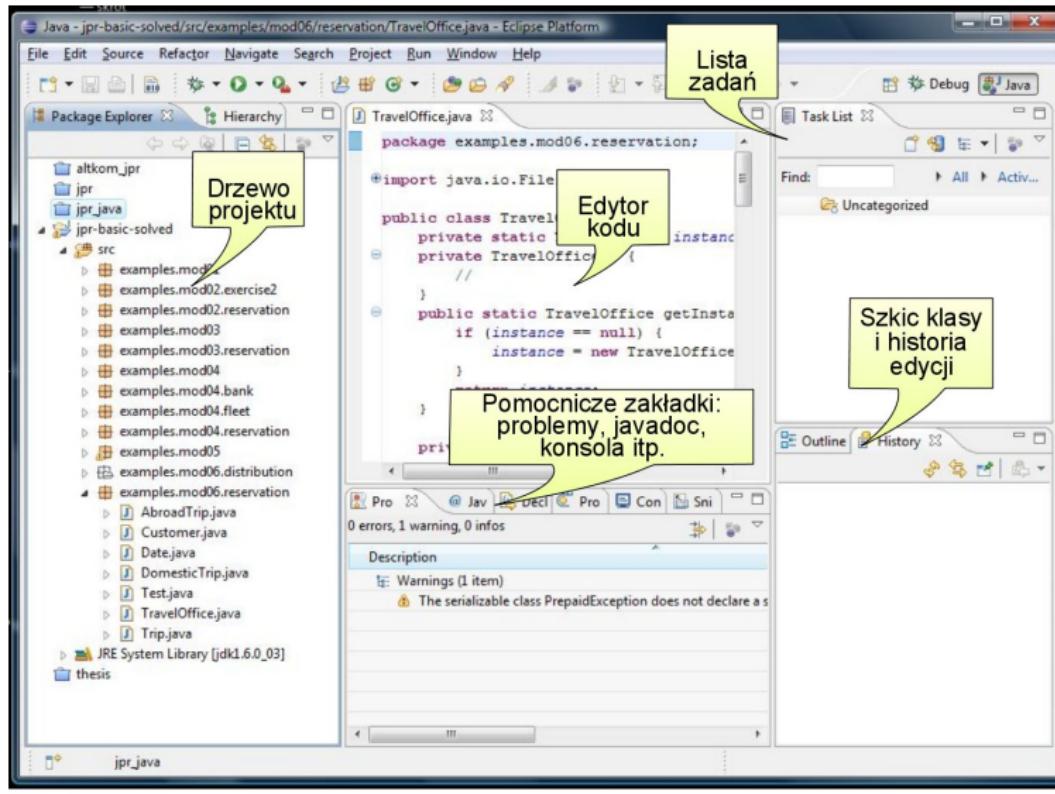
Typowe cechy środowiska pracy

- Bogaty edytor kodu – uzupełnianie składni, oznaczanie błędów i ostrzeżeń, podpowiadanie sposobów usunięcia błędów, refactoring, automaty do generowania kodu
- Automatyczne budowanie projektu
- Możliwość szybkiego uruchomienia projektu
- Szybkie przemieszczanie się po projekcie
- Tworzenie *javadoc*, *jar* za pomocą prostych kreatorów
- ... i wiele innych w zależności od produktu

Eclipse IDE – organizacja miejsca pracy

- Działa na bazie wtyczek
- Jeden z najlepiej dopracowanych edytorów kodu źródłowego
- Dla programistów, którzy nie chcą polegać tylko i wyłącznie na automagicznym kodzie generowanym przez kreatory
- **Szybki**
- <http://www.eclipse.org/downloads/>

Eclipse IDE – organizacja miejsca pracy

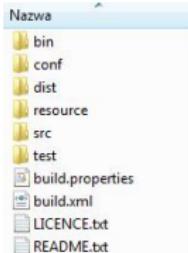


Struktura projektu

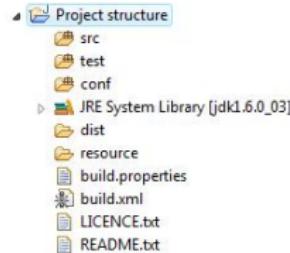
- Pozwala łatwiej orientować się w złożonym projekcie
- Porządkuje projekt
- Upraszczza szukanie przyczyn błędów
- Podstawowe aspekty:
 - Struktura **katalogów** w projekcie
 - Struktura **pakietów**

Struktura katalogów

- Służy do logicznego wydzielenia części projektów: kodu źródłowego, testów, plików konfiguracyjnych, plików zasobów



Natynka struktura plików



Ta sama struktura zarządzana przez Eclipse

- src – kod źródłowy aplikacji
- resource – pliki zasobów (obrazki, pliki tekstowe)
- test – kod źródłowy testów
- conf – pliki konfiguracyjne

- bin – skompilowane klasy i zasoby niezbędne do ich działania
- dist – katalog z wersją dystrybucyjną aplikacji
- build.xml, build.properties – konfiguracja budowania projektu za pomocą narzędzia Ant



Przykłady innych struktur katalogów

- Istnieje wiele alternatywnych sposobów organizacji projektów
- W projekcie należy dostosować strukturę do potrzeb
- Inne przykłady:
 - konwencje Sun:
[http://java.sun.com/blueprints/code/
projectconventions.html](http://java.sun.com/blueprints/code/projectconventions.html)
 - konwencje związane z systemem budowania Maven:
[http://maven.apache.org/guides/introduction/
introduction-to-the-standard-directory-layout.html](http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html)

Struktura pakietów

- Pakiety są zasadniczym elementem porządkującym klasy w Javie
- Służą do logicznego grupowania powiązanych ze sobą klas
- W miarę możliwości należy minimalizować zależność klas w danym pakiecie od klas w innych pakietach
- Najczęściej główny pakiet danego projektu ma taką konstrukcję:

<nazwa-lub-url-firmy>.<nazwa-projektu>

- Przykład:

pl.altkom.crm

Struktura pakietów

- W dalszej kolejności podziału pakietów powinny być moduły lub funkcjonalności udostępniane przez aplikację lub bibliotekę, np.:

pl.altkom.crm.accounts
pl.altkom.crm.calendar
pl.altkom.crm.campaigns

-
- Jeśli aplikacja jest budowana w architekturze dwu- lub trójwarstwowej, kolejnym stopniem podziału mogą być warstwy, np.:

pl.altkom.crm.campagins.model
pl.altkom.crm.campagins.services
pl.altkom.crm.campagins.gui

Struktura pakietów

- Czasami wydziela się pakiet dla wyjątków w danym module, np.:

```
pl.altkom.crm.accounts.exceptions
```

- Jeśli aplikacja budowana jest w oparciu o silne użycie interfejsów wydziela się specjalny pakiet grupujący implementacje tych interfejsów, np.:

```
pl.altkom.crm.accounts.services.impl
```

- Tworząc nowe pakiety należy tak budować strukturę, aby pakiety tworzyły katalog informacji o projekcie

Klasa z metodą *main*

- Realizując samodzielnie projekt oparty o Javę SE musimy mieć jedną klasę, dzięki której projekt będzie mógł zostać uruchomiony
- Klasa ta musi zawierać metodę *main*
- Klasa ta powinna zawierać kod inicjalizujący wszystkie elementy niezbędne do działania aplikacji (np. wczytanie konfiguracji, pobranie danych z sieci, uruchomienie puli połączeń z bazą danych)
- Wszelkie konkretne operacje powinny być **delegowane** do klas odpowiedzialnych za poszczególne moduły aplikacji

Podsumowanie

- Środowisko pracy – Eclipse IDE
- Struktura projektu
- Organizacja pakietów oraz klasa główna

Pytania sprawdzające

- W czym może być pomocny program Eclipse?
- Jakie jest przeznaczenie podstawowych okien Eclipse'a?
- Podaj przykładową strukturę pakietów
- Która metoda powinna zawierać kod inicjalizujący wszystkie elementy niezbędne do działania aplikacji?

Plan szkolenia

- 1 Wprowadzenie do technologii Java
- 2 Proste obiekty i typy danych
- 3 Zasady budowania aplikacji w Javie
- 4 Podstawy składni języka**
- 5 Koncepcje programowania obiektowego
- 6 Obsługa błędów i wyjątków
- 7 Środowisko systemowe i operacje we/wy
- 8 Najczęściej popełniane błędy w Javie

Plan modułu

4

Podstawy składni języka

- identyfikatory i konwencje nazewnicze
- operacje na zmiennych
- instrukcje sterujące
- tablice

Identyfikatory

- **Identyfikator**, czyli nazwa zmiennej, klasy, metody
 - może mieć dowolną długość i składać się z dowolnych znaków alfanumerycznych, pod następującymi warunkami:
 - pierwszy znak jest literą lub znakiem \$ lub _
 - identyfikator nie jest tożsamy ze słowem kluczowym
- Przykłady:

```
value, param1, blad, thisMan, _param, $money$
```

- Litery są znakami w Unicode
 - mogą zawierać znaki narodowe
- Wielkość liter w nazwach ma znaczenie

Konwencja nazewnictwa

- Zebrane w dokumencie
<http://java.sun.com/docs/codeconv/>
- Jej stosowanie poprawia czytelność kodu
- Stosowanie nie jest obowiązkowe, ale zalecane
- Konwencja jest stosowana przez większość programistów
- Niektóre rozwiązania technologiczne wymagają zgodności z konwencją (np. nazewnictwo metod w *JavaBeans*)

Konwencja nazewnictwa

- Nazwy klas

- forma rzeczownikowa
- nazwy rozpoczynane są dużą literą
- jeśli nazwa składa się z kilku wyrazów, każdy z nich jest rozpoczynany dużą literą

- Przykłady:

String, Object, Toolkit, Connection,
DriverManager, HashSet, ArrayList

Konwencja nazewnictwa

• Nazwy metod

- forma czasownikowa
- nazwy rozpoczynane są małą literą
- jeśli nazwa składa się z kilku wyrazów, każdy następny rozpoczynany jest dużą literą

```
assignTripToCustomer, withdrawMoney
```

Konwencja nazewnictwa

• Nazwy metod w JavaBeans

- metody ustawiające wartość atrybutu (*setters*) rozpoczynają się słowem *set*

`setName, setPhoneNumber, setEndFlag`

-
- metody odczytujące wartość atrybutu (*getters*) rozpoczynają się słowem *get* (*is* dla atrybutów typu *boolean*)

`getBalance, getMaxNumber, isAdmin`

Konwencja nazewnictwa

- Nazwy zmiennych i atrybutów
 - forma rzeczownikowa
 - nazwy rozpoczynane są małą literą
 - jeśli nazwa składa się z kilku wyrazów, każdy z nich jest rozpoczynany dużą literą
 - dopuszczalne jest stosowanie jednoliterowych nazw jako liczników pętli
- Przykłady:

`counter, currentTime, bigValue`

- Nazwy zmiennych statycznych pisane są dużymi literami
 - wyrazy rozdzielane są znakiem podkreślenia

`MAX_VALUE, TIME_LIMIT`

Formatowanie kodu źródłowego

- Długość wiersza 80 znaków
- Łamanie wierszy
 - po przecinku, przed operatorem
 - jeśli to możliwe, zachowuj w całości wyrażenia (np. wywołania funkcji)
 - jeśli wyrażenie jest zbyt skomplikowane, rozważ wprowadzenie lokalnych zmiennych pomocniczych
 - wcięcia w kodzie powinny odzwierciedlać logikę algorytmu
- Przykład:

```
if ((condition1 && condition2)
    || (condition3 && condition4)
    || !(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Konwencje kodowania – narzędzia

- Istnieją narzędzia, które pozwalają kontrolować zgodność z konwencją kodowania
- Środowiska programistyczne (IDE) często posiadają narzędzia do badania podstawowych reguł
- Pełne sprawdzanie można sobie zapewnić narzędziami, które są w pełni konfigurowalne
- Checkstyle <http://checkstyle.sourceforge.net/>
- Również w formie pluginów do środowisk programistycznych

Operacje na zmiennych

- Operatory służą do wykonywania operacji:
 - arytmetycznych
 - logicznych
 - przekształceń bitowych
 - porównań
 - przypisywania wartości

Operacje arytmetyczne

- Operacje arytmetyczne wykonywane poprzez operatory:

mnożenie:	$x = x * y;$
dzielenie:	$x = x / y;$
reszta z dzielenia całkowitego:	$x = x \% y;$
dodawanie:	$x = x + y;$
odejmowanie:	$x = x - y;$
inkrementacja zmiennej:	$++x;$ lub $x++;$
dekrementacja zmiennej:	$--x;$ lub $x--;$
nadanie wartości x wyniku operacji z y :	$x [op] = y;$
równoważne: gdzie $[op]$ jest jednym z operatorów:	$x = x [op] y;$ $+, -, *, /, \%$

Operacje relacyjne

- Porównania liczb:

większy:	$x > y$
większy lub równy:	$x \geq y$
mniejszy:	$x < y$
mniejszy lub równy:	$x \leq y$
równy:	$x == y$
różny:	$x != y$

Operacje logiczne

- Operator logiczny

Negacja:	$\neg x$
Operacja AND:	$x \text{ && } y$
Operacja OR:	$x \text{ } y$

- Operatory `&&` i `||` gwarantują, że przetwarzanie wyrażenia zostanie przerwane, gdy znany już będzie wynik logiczny, np.:

```
if (x != null && x.isValid()) {}
```

```
if (x == null || x.isValid()) {}
```

Operacje binarne

- Wykonywane są na liczbach całkowitych
 - operacja AND: $x \And y$

$01001011 \And 00001111 = 00001011$

-
- operacja OR: $x \Or y$

$01100011 \Or 11000000 = 11100011$

-
- operacja XOR: $x \Xor y$

$00110011 \Xor 01100110 = 01010101$

Operacje binarne

- Wykonywane są na liczbach całkowitych
 - przesunięcie w lewo: $x \ll y$

$00101010 \ll 1 = 01010100$

$$32 \ll 1 = [32 * 2] = 64$$

- przesunięcie w prawo z uzupełnieniem bitami znaku: $x \gg y$

$00101010 \gg 1 = 00010101$

$$32 \gg 1 = [32 / 2] = 16$$

Operacje binarne

- Wykonywane są na liczbach całkowitych
 - przesunięcie w prawo z uzupełnieniem zerami: $x >>> y$

$101010 >>> 1 = 010101$

Różnica w działaniu operatora $>>$ w stosunku do operatora $>>>$ jest widoczna dla liczb ujemnych

- negacja logiczna: $\sim x$

$\sim 101010 = 010101$

Kolejność wykonywania operacji

● Kolejność wykonywania operacji

() . []
(rzutowanie) ! ~ ++ --
* / %
+ -
<< >> >>>
< <= > >=
== !=
&
^
|
&&
||
? :
= += -= *= %= itp.

Kolejność wykonywania operacji

- W zapamiętaniu kolejności operatorów mogą pomóc poniższe mnemoniki:

Jedzie	(jednoargumentowe)	Ulcer	(unary)
Andrzej	(arytmetyczne)	addicts	(arithmetic)
rowerem	(relacyjne)	really	(relational)
Iedwie	(logiczne)	like	(logical)
wlokąc	(warunkowe)	C	(conditional)
pedałami	(przypisania)	a lot	(assignment)

Instrukcja warunkowa

```
if (wyrazenie_logiczne) {  
    // operacje  
}
```

lub:

```
if (wyrazenie_logiczne) {  
    /* operacje wykonywane, jesli wyrazenie_logiczne jest  
     * prawdziwe */  
} else {  
    /* operacje wykonywane, jesli wyrazenie_logiczne jest  
     * falszywe */  
}
```

Przykład instrukcji warunkowej

```
if (command.equals("OK")) {  
    // make OK action  
} else if (command.equals("Cancel")) {  
    // make Cancel action  
} else if (command.equals("Add")) {  
    // make Add action  
} else {  
    // ignore  
}
```

Pętla for

Składnia:

```
for (wyr_począt; war_logiczny; wyr_petli) {  
    // blok operacji  
}
```

Przykład:

```
for (int i = 0; i < vector.size(); i++) {  
    vector.removeElementAt(i);  
}
```

Nowa konstrukcja pętli `for`

- Pętla ta jest nazywana pętlą *for-each*
- Dostępna jest od Javy SE 5.0
- Składnia:

```
for (Object i : Collection<Object>) {  
    // blok operacji  
}
```

- Przykład:

```
int tab[] = {1, 2, 3, 4};  
int sum = 0;  
for (int i : tab) {  
    sum += i;  
}
```

Pętla while

- Składnia:

```
while (wyrazenie_logiczne) {  
    // blok operacji  
}
```

- Przykład:

```
while (enum12.hasMoreElements()) {  
    System.out.println(enum12.nextElement());  
}
```

Pętla do . . . while

- Składnia:

```
do {  
    // blok operacji  
} while (wyrażenie_logiczne);
```

- Przykład:

```
do {  
    query.showNextPage();  
} while(query.hasMore());
```

Instrukcje break/continue

- Realizacja instrukcji w bloku pętli może zostać przerwana
 - break* – przerywa pętlę
 - continue* – rozpoczyna nowy przebieg pętli
- Przykład:

```
while (i < MAX_LIMIT) {  
    // operacje  
    if (i%100 == 0) {  
        continue;  
    }  
    // operacje  
}
```

Instrukcja switch

- Składnia:

```
switch (licz_calkowita) {  
    case wartosc1:  
        operacje;  
    case wartosc2:  
        operacje;  
    default:  
        operacje;  
}
```

- Argumentem instrukcji *switch* musi być wartość typu: *int*, *short*, *byte*, *char* lub *enum* (od JSE 7 – także *String*!)

Instrukcja switch

- Przykład:

```
switch (result) {  
    case ADD:  
        add();  
        break;  
    case REMOVE:  
        remove();  
        break;  
    default:  
        show();  
}
```

- Należy pamiętać o instrukcji *break* w kodzie, aby zakończyć sterowanie

Tablice

- Tablica jest grupą zmiennych o tym samym typie
- Zmienna tablicowa jest zmienną referencyjną
- Tablica jest obiektem
- Deklaracja zmiennej tablicowej

```
int [] tab, word;  
Date d[];
```

- deklaracja zmiennej nie tworzy tablicy
- Nie można zmienić rozmiaru tablicy

Tablice

- Tablica, tak jak i inne obiekty, jest tworzona przy pomocy operatora *new*

```
tab = new int[10];  
d = new Date[5];
```

- Indeksy tablicy: 0 ÷ n-1

```
tab[5] = 1998;
```

- Kontrola indeksów tablicy jest prowadzona przez JVM
- Kopiowanie zmiennej tablicowej nie kopiuje tablicy

```
int tab2[] = tab; // powielona jest tylko referencja
```

Tablice

- Tablica dla typów prostych przechowuje ich wartości
- Tablica dla typów referencyjnych przechowuje referencje do obiektów

```
Date d[] = new Date[5];  
d[0] = new Date(1999, 10, 12);  
d[1] = new Date(1000, 10, 10);
```

- Tablica typów referencyjnych pozwala na umieszczanie referencji do obiektów tylko zadeklarowanego typu

```
d[2] = new String("kot"); // zle
```

Inicjalizacja tablic

- Tablica może być wstępnie wypełniona elementami

```
String msg[] = {"ala", "kot", "pies"};
int tab[] = {1, 2, 3, 4, 5};
Date d[] = {new Date(1999, 10, 12),
            new Date(1000, 10, 12),
            new Date(2004, 10, 12)};
```

-
- Dla inicjalizowanych tablic nie określa się rozmiaru
 - Rozmiar jest wyliczany na podstawie danych

Przeglądanie tablic

- Tablica, jako obiekt, posiada jeden atrybut określający jej długość (liczbę elementów)
- Przeglądanie tablic powinno opierać się o wartość atrybutu *length*

```
int tab[] = new int[5];
for (int i = 0; i < tab.length; i++) {
    tab[i] = i * i;
}
```

- Tablica może być efektywnie kopowana poprzez:

```
System.arraycopy(src, od_el, dest, od_el, ile);
```

Tablice wielowymiarowe

- Tablica tablic staje się tablicą dwuwymiarową
- W podobny sposób można tworzyć tablice o większej liczbie wymiarów
- Deklaracja tablicy dwuwymiarowej:

```
char [][] tab;  
char []tab[];  
char tab[][];
```

- Każdy z wymiarów ma własny atrybut *length*

Tablice wielowymiarowe

- Przykład deklaracji:

```
int tab[][] = new int[5][10];
Date d[][] = new Date[2][];
d[0] = new Date[6];
d[1] = new Date[8];

/* pierwszy wymiar tablicy musi byc podany */
String msg[][] = new String[][][5]; // zle
```

- tablica *tab* to typowa macierz dwuwymiarowa
- tablica *d* to tzw. tablica postrzępiona (ang. *jagged array*)

Tablice wielowymiarowe

- Wstępna inicjalizacja tablicy dwuwymiarowej:

```
String msg [][] = { {"ala", "ola", "ela"},  
                     {"kot", "pies"} };
```

- Przeglądanie tablicy dwuwymiarowej:

```
for (int i = 0; i < msg.length; i++) {  
    for (int j = 0; j < msg[i].length; j++) {  
        System.out.println(msg[i][j]);  
    }  
}
```

- Zadanie: powyższy przykład zaimplementuj używając zagnieżdżonej pętli “for-each”

Podsumowanie

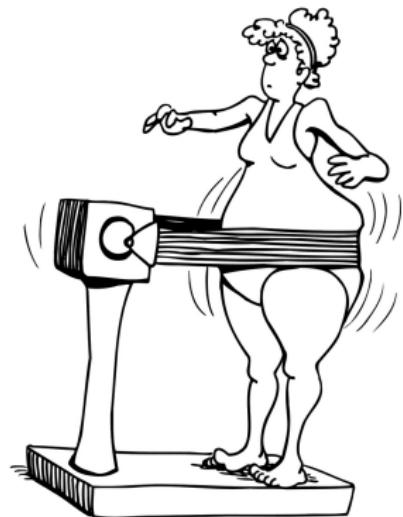
- Identyfikatory
 - konwencje nazewnictwa
- Operacje na zmiennych
 - rodzaje operatorów
- Instrukcje sterujące
 - *for, while, do-while, switch*
- Tablice

Pytania sprawdzające

- Jakie warunki spełnia poprawny identyfikator w Javie?
- Podaj zasady konwencji nazewniczej dla klas, metod i atrybutów
- Podaj przykłady operatorów relacyjnych i logicznych
- Jaka jest konstrukcja instrukcji warunkowej?
- Wymień rodzaje pętli w Javie
- Jak można przedwcześnie zakończyć pętlę?
- Jaką składnię ma instrukcja *switch*?
- Jak odczytać rozmiar zadeklarowanej tablicy?
- Jak są numerowane elementy tablicy?
- Co się stanie gdy odwołamy się do elementu tablicy poprzez nieprawidłowy indeks?
- Jak można skopiować zawartość jednej tablicy do innej?
- Jak utworzyć tablicę dwuwymiarową w której każdy wiersz będzie miał inną liczbę elementów?

Ćwiczenia

- Ćwiczenie 4.1:
pętla, instrukcje warunkowe
- Ćwiczenie 4.2:
tablice, kopiowanie danych
- Ćwiczenie 4.3:
*pętla, tablica, korzystanie
z dokumentacji*
- Ćwiczenie 4.4:
*TravelOffice – dynamiczna “kolekcja
danych”*



Plan szkolenia

- 1 Wprowadzenie do technologii Java
- 2 Proste obiekty i typy danych
- 3 Zasady budowania aplikacji w Javie
- 4 Podstawy składni języka
- 5 Koncepcje programowania obiektowego**
- 6 Obsługa błędów i wyjątków
- 7 Środowisko systemowe i operacje we/wy
- 8 Najczęściej popełniane błędy w Javie

Plan modułu

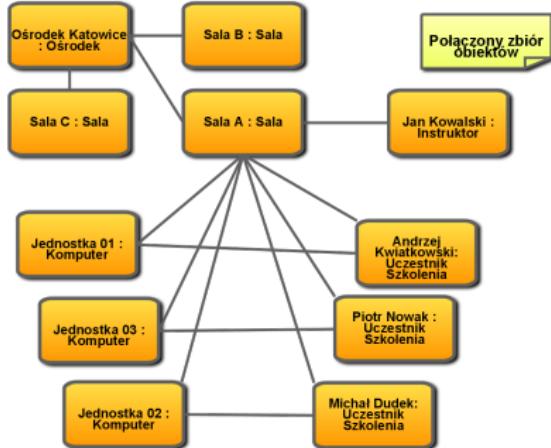
5

Koncepcje programowania obiektowego

- modelowanie obiektowe
- podstawowe pojęcia obiektowe
- tworzenie obiektu
- klasa Object
- elementy statyczne
- klasy abstrakcyjne
- interfejsy
- klasy wewnętrzne
- jakość modelu obiektowego

Wszystko jest obiektem

- Systemy informatyczne tworzone zgodnie z paradigmatem obiektowym to świat zbudowany z klocków (zwanych obiektami) powiązanych i współpracujących ze sobą



Wprowadzenie do obiektów

- Obiekty to pojęcia charakteryzujące się stanem i określonym zachowaniem
- Obiekty wchodzą ze sobą w relacje, dzięki czemu pozwalają tworzyć większe, bardziej złożone pojęcia lub systemy
- Modelowanie obiektowe to poszukiwanie odpowiednich definicji obiektów i relacji między nimi
- Dla każdego obiektu określa się:
 - jego odpowiedzialność w systemie (jaka jest jego rola, co może zrobić dla innych obiektów)
 - z kim będzie współpracować w celu osiągnięcia określonego celu

Przykłady obiektów

Klasa	Odpowiedzialność	Współpraca
<i>String</i>	pozwala przechowywać i przetwarzaćłańcuchy uporządkowanych znaków	dowolne obiekty korzystające złańcuchów znaków
<i>Connection</i>	reprezentuje otwarte połączenie do bazy danych, umożliwiające zadawanie zapytań	<i>DriverManager</i> ; obiekty pozwalające na wydawanie zapytań: <i>Statement</i> , <i>PreparedStatement</i> , <i>CallableStatement</i>
<i>Animal</i>	opisuje zwierzę i wybrane elementy wyglądu	dowolne obiekty przetwarzające informacje o zwierzętach
<i>Window</i>	reprezentuje okno bez paska tytułu, które może uzyskać reprezentację graficzną na ekranie komputera; stanowi kontener dla komponentów graficznych	dowolne komponenty graficzne oraz klasy systemowe

Podejście obiektowe

- System jest organizowany poprzez współpracujące ze sobą obiekty
- Obiekty między sobą przesyłają komunikaty (wołają metody)
- Obiekt wiąże w sobie dane i zachowanie/operacje na tych danych
- Modelowanie skupia się na określeniu sposobu współpracy i odpowiedzialności między obiektami, a nie na implementacji
 - model obiektów przedstawiony graficznie jest łatwiejszy do przyswojenia i zrozumienia
 - model graficzny ewoluje w stronę coraz bardziej szczegółowego przedstawienia systemu
 - model jest wprowadzeniem do implementacji

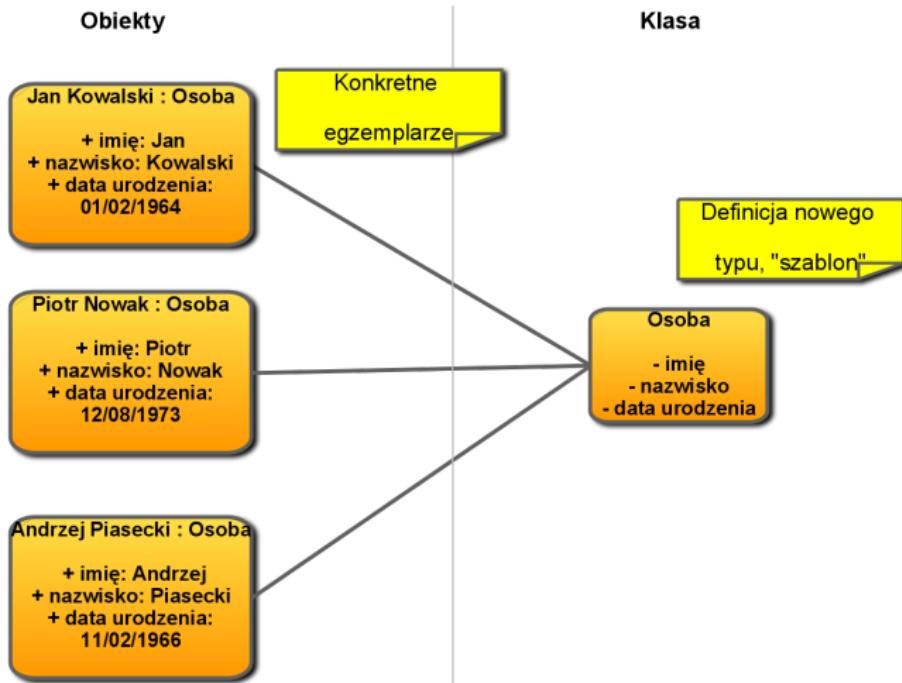
Klasy i obiekty

- Klasa stanowi ogólną reprezentację danego typu
- Obiekt jest indywidualnym egzemplarzem utworzonym na podstawie definicji klasy

KLASA	OBIEKTY
Ludzie	konkretnie osoby, np. Jurek, Janek, Ola
Samochody	konkretnie auta o określonym numerze rejestracyjnym

- Klasa stanowi "formę", z której wytwarzane są "cegiełki" – obiekty
- Z danej klasy powstaje wiele obiektów o tych samych atrybutach, ale różniących się między sobą zawartością danych (czyli stanem)
 - **Klasa:** Dog {breed; name}
 - **Obiekt:** dog1 {beagle, snoopy}; dog2 {golden, molly}

Klasy



Główne pojęcia obiektowe

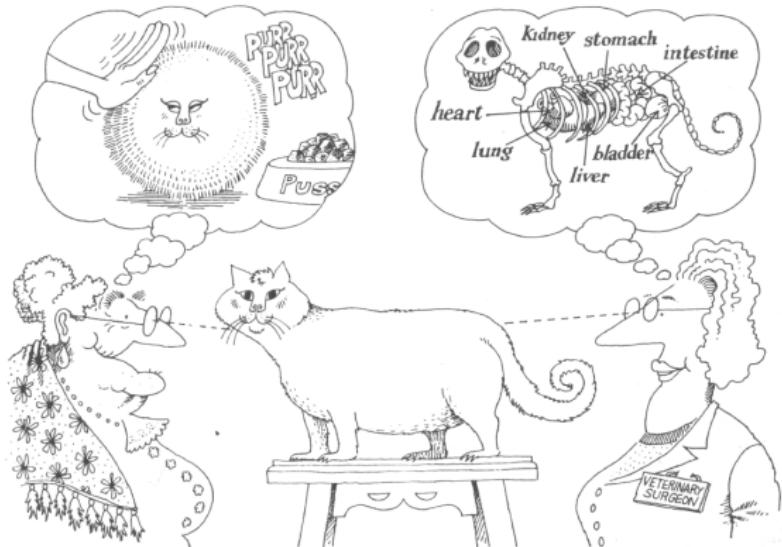
- Abstrakcja danych
- Hermetyzacja danych
- Asocjacja
- Agregacja
- Kompozycja
- Dziedziczenie – specjalizacja/generalizacja
- Polimorfizm
- Spójność i sprzężenie obiektów

Abstrakcja danych

- Abstrakcja danych umożliwia odpowiednią generalizację modelu
 - upraszcza rzeczywistość – pozwala na ignorowanie zbędnych detali
- Dzięki abstrakcji (uogólnianiu pojęć) modelowanie złożonych systemów staje się prostsze
 - wizualizujemy lub modelujemy na takim poziomie ogólności, jaki jest w danym momencie niezbędny
 - uogólnione typy są następnie uszczegóławiane zgodnie z potrzebami

Abstrakcja danych

The First Section: Concepts



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Hermetyzacja danych

- Hermetyzacja (*encapsulation*) – ukrywanie danych ma na celu osłonięcie danych obiektu przed bezpośrednią zmianą:

```
public class Date {  
    int year, month, day;  
}
```

```
Date d = new Date();  
d.month = 2;  
d.day = 30;
```

- Dane, które nie są chronione łatwo doprowadzić do niespójności

Hermetyzacja danych

- Ukrycie danych pozwala na wprowadzenie kontroli zmian ich wartości poprzez kod metody:

```
public class Date {  
    private int year, month, day;  
    // ...  
    public boolean setDay(int d) {  
        if (d >31) {  
            return false;  
        } else if (...) { ... }  
    }  
}
```

```
Date d = new Date();  
d.day = 32; // atrybut nie jest dostepny !!!  
d.setDay(32); // kontrola poprawnosci danych w metodzie !!!
```

Hermetyzacja danych

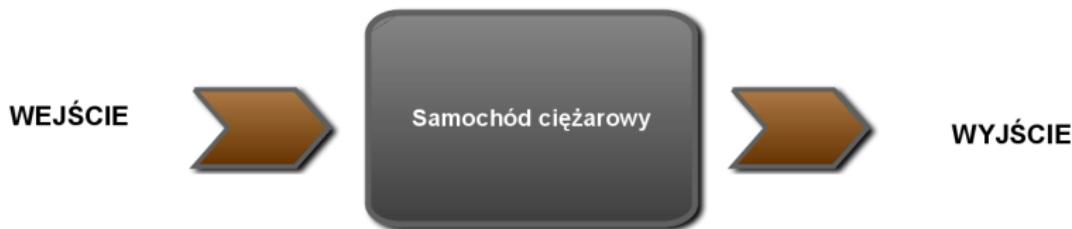
- Ukrycie danych pozwala na zmianę wewnętrznej reprezentacji danych bez konieczności zmiany sposobu korzystania z danego typu

```
public class Date {  
    private long time;  
    public boolean setDay(int day) { /* odpowiedni kod */ }  
    public boolean setMonth(int month) { /* odpowiedni kod */ }  
    public boolean setYear(int year) { /* odpowiedni kod */ }  
}
```

```
Date d = new Date();  
d.setDay(30);  
d.setMonth(1);
```

Hermetyzacja danych

Klasę traktujemy jak czarną skrzynkę. Ukrywamy szczegóły implementacji (m. in. pola klasy)



Relacje między obiekta

Relacje określają jaką rolę pełnią obiekty w danym związk

- Asocjacja

- jeden obiekt korzysta z usług drugiego
- Driver-Car: oba obiekty współpracują, aby zrealizować cel – przemieścić samochód



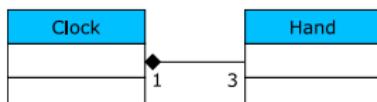
- Agregacja (rozwinięcie asocjacji)

- jeden obiekt uzupełnia się innym obiektem lub zawiera kolekcję innych obiektów
- Group-User: grupa przechowuje kolekcję użytkowników



- Kompozycja (rozwinięcie agregacji)

- jeden z obiektów zawiera w sobie lub składa się z innych obiektów (bez nich nie mógłby istnieć)
- Clock-Hand: zegar wchodzi w kompozycję ze wskaźówkami



Relacje między obiekta

- Implementacja relacji

- Asocjacja

- referencja do klasy asocjowanej jest oczekiwana tylko w czasie realizacji wspólnego zadania
 - oba obiekty żyją niezależnie od siebie

- Agregacja

- referencja do obiektów agregowanych jest tworzona zazwyczaj na początku życia obiektu głównego
 - operacje obiektu głównego są w dużej mierze realizowane na obiektach agregowanych

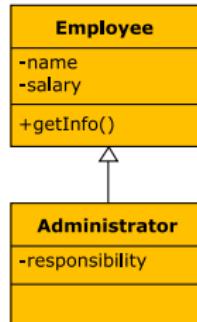
- Kompozycja

- obiekty komponowane są tworzone wewnątrz obiektu głównego (np. w konstruktorze)
 - czas życia obiektów komponowanych nigdy nie przekracza czasu życia obiektu głównego

Dziedziczenie

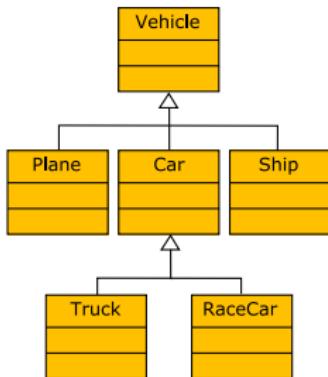
- Umożliwia zbudowanie nowej klasy korzystającej z własności klasy już istniejącej
- Klasa zbudowana na bazie klasy już istniejącej "dziedziczy" jej zachowanie (a zarazem rodzaj przechowywanych danych)

```
public class Employee {  
    String name;  
    int salary;  
    public String getInfo() { /* ... */ }  
}  
  
public class Administrator extends Employee {  
    String responsibility;  
}
```



Dziedziczenie

- Dziedziczenie jest możliwe tylko z jednej klasy bazowej
 - dziedziczenie – to inaczej relacja generalizacji-specjalizacji
 - grot strzałki w notacji UML wskazuje kierunek generalizacji
 - klasa potomna specjalizuje (uszczegóławia) klasę bardziej ogólną



Dziedziczenie i konstruktor

- Klasa potomna (powstała na bazie już istniejącej) korzysta z dostępnych atrybutów i metod klasy bazowej
- Klasa potomna nie dziedziczy konstruktorów klasy bazowej
- Wskazanie klasy bazowej odbywa się po słowie kluczowym **extends**

```
public class Dog extends Animal {  
    private String petName;  
    public void bark() {  
        /* ... */  
    }  
}
```

Dziedziczenie i konstruktor

- Konstruktor nie jest dziedziczony
- Klasa potomna z konstruktorem domyślnym oczekuje istnienia w klasie bazowej konstruktora bezargumentowego
 - jeśli takiego nie ma, klasa potomna musi jawnie wywołać jeden z dostępnych konstruktorów klasy bazowej
- Wywołanie konstruktora z klasy bazowej musi nastąpić w pierwszej znaczącej linii konstruktora klasy potomnej
- Konstruktor z klasy bazowej wywołujemy poprzez użycie słowa kluczowego **super**
- Konstruktor domyślny zawsze wywołuje konstruktor bezargumentowy

Dziedziczenie i konstruktor

```
public class Animal {  
    String name;  
    public Animal(String name) {  
        this.name = name;  
    }  
}  
  
public class Dog extends Animal {  
    public Dog(String name) {  
        super(name); // wywolanie konstruktora  
        // reszta kodu  
    }  
}
```

Modyfikatory dostępu

Modyfikator	Ta sama klasa	Ten sam pakiet	W relacji dziedziczenia	Wszyscy
private	x			
<pusty>	x	x		
protected	x	x	x	
public	x	x	x	x

Przeddefiniowanie metod

- Klasa potomna może dostarczyć nowej definicji metody dostępnej w klasie bazowej (*override*)
- Przedefiniowywana metoda musi mieć takie same co metoda z klasy bazowej:
 - nazwę
 - listę argumentów
 - typ zwracanej wartości
- Przedefiniowywana metoda nie może oferować węższego dostępu niż metoda klasy bazowej
- Dostęp do metody z klasy bazowej można uzyskać poprzez referencję ***super***

Przeddefiniowanie metod – przykład

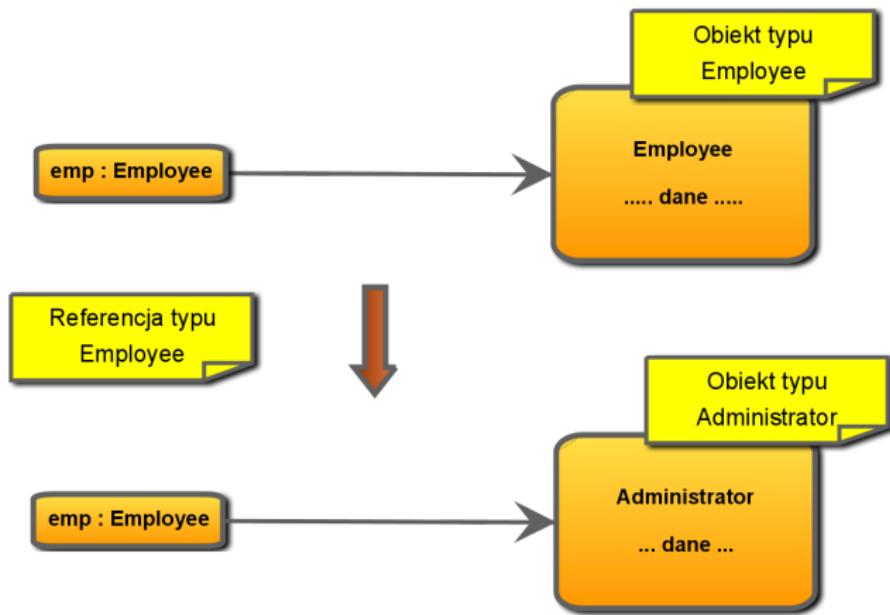
```
public class Employee {  
    String name;  
    int salary;  
    public String getInfo() {  
        return name + ", salary: " + salary;  
    }  
}  
  
public class Administrator extends Employee {  
    String responsibility;  
  
    // przeddefiniowanie metody z klasy bazowej  
    @Override  
    public String getInfo() {  
        return super.getInfo() + ", resp: " + responsibility;  
    }  
}
```

Polimorfizm

- **Polimorfizm**, czyli **wielopostaciowość** oznacza, że dana referencja może mieć dostęp do wielu różnych form obiektów
 - jeśli referencja może odnosić się do wielu obiektów, to tym samym zachowanie się obiektu dostępnego przez referencje wydaje się różne
- Na przykład:
 - z relacji dziedziczenia wynika, że *Administrator* jest równocześnie obiektem typu *Employee*
 - to oznacza, że referencja typu *Employee* może odnosić się również do obiektów typu *Administrator*

```
Employee emp = new Employee();
emp.getInfo(); // wywolanie metody z Employee
emp = new Administrator();
emp.getInfo(); // wywolanie metody z Administrator
```

Polimorfizm

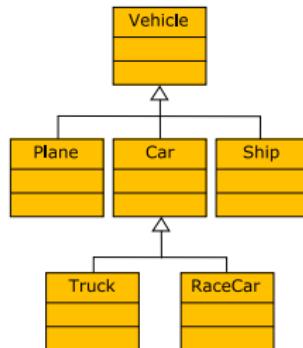


Kolekcje heterogeniczne

- Dzięki istnieniu relacji między klasą bazową, a klasą potomną możliwe jest tworzenie kolekcji obiektów opartych o wspólny typ

```
Vehicle v[] = new Vehicle[4];
v[0] = new Plane();
v[1] = new Ship();
v[2] = new Truck();
v[3] = new RaceCar();

for (int i = 0; i < v.length; i++) {
    v[i].start();
}
```



- w powyższym przykładzie implementacja metody *start* może być w każdym konkretnym typie inaczej zrealizowana
- Posiadanie wspólnego przodka umożliwia zarządzanie kolekcją (wydawanie poleceń) w spójny sposób

Konwersja typów

- Referencja typu bazowego może służyć do odnoszenia się również do obiektów potomnych

```
Vehicle v = new RaceCar();
```

- Referencja typu potomnego potencjalnie może przyjąć wartość referencji typu bazowego
 - pod warunkiem, że rzeczywisty obiekt na który wskazuje referencja typu bazowego jest obiektem zgodnym z referencją typu potomnego
 - operacja przypisania musi być poprzedzona konwersją typów

```
Car c = (Car) v;
```

Konwersja typów

- Konwersja referencji typu potomnego na bazowy zachodzi automatycznie
- Konwersja referencji typu bazowego na potomny musi być wymuszana, bo nie zawsze jest możliwa
- Konwersja typów z dwóch gałęzi hierarchii nigdy nie jest możliwa

```
Plane p = new Plane();  
Car c = (Car) p;
```

Konwersja typów

- W przypadku konwersji typu potomnego na bazowy sprawdzenie poprawności zachodzi w trakcie wykonywania kodu
- Aby zapewnić poprawność wykonania konwersji należy sprawdzać możliwość jej wykonania przy pomocy operatora *instanceof*

```
Vehicle vehicle = new RaceCar();  
if (vehicle instanceof Car) {  
    Car car = (Car) vehicle;  
    car.wiperOn();  
}
```

- Typ bazowy jeśli odnosi się do obiektu typu potomnego nie ma dostępu do jego specyficznych metod
 - należy dokonać konwersji

Argumenty metod

- Argument przekazywany do metody może korzystać z zachowania polimorficznego

```
public String prepareReport (Vehicle v) {  
    // ...  
    v.getInfo();  
    // ...  
}
```

```
Vehicle v[] = new Vehicle[2];  
v[0] = new Plane();  
v[1] = new RaceCar();  
for (int i = 0; i < v.length; i++) {  
    prepareReport(v[i]);  
}
```

Inicjacja obiektu

- Utworzenie obiektu poprzez operator *new* powoduje:
 - alokację pamięci na obiekt
 - nadanie zmiennym klasy wartości domyślnych
 - wywołanie konstruktora klasy
 - podstawienie wartości pod argumenty konstruktora
 - wywołanie konstruktora klasy bazowej (*super*) lub innego konstruktora tej samej klasy (*this*), który z kolei wywoła konstruktor klasy bazowej
 - inicjacja obiektu bazowego zachodzi w analogiczny sposób aż do poziomu obiektu *Object*
 - inicjacja zmiennych klasy podanymi wartościami
 - wykonanie instrukcji konstruktora
 - zwrócenie referencji do obiektu

Inicjacja obiektu

Wnioski

- konstruktor nie powinien wołać metod, które mogą być przedefiniowane
 - metoda polimorficzna wywołana z poziomu konstruktora klasy bazowej, będzie zrealizowana na danych klasy potomnej, które nie zostały jeszcze w pełni zainicjowane
- jeśli konstruktor wywołuje metody, to powinny one być oznaczone jako prywatne

Object

- Każda klasa ma wspólnego przodka – *Object*
- Jeśli klasa w sposób jawnny nie dziedziczy po innej klasie, to dziedziczy po *Object*
- Każda klasa przejawia więc zachowanie z klasy *Object*, a zatem posiada m.in. metody:
 - *toString()* – zwraca reprezentację tekstową obiektu
 - *equals(Object o)* – porównuje dany obiekt do podanego argumentu
 - *hashCode()* – zwraca w miarę unikatową wartość typu *int* zależną od stanu obiektu – metoda używana jako pomoc w identyfikacji obiektów
- Kolekcje obiektów można opierać o wspólnego rodzica jakim jest *Object*
 - wadą jest brak metod charakteryzujących daną rodzinę
 - konieczność dokonywania konwersji typów

Object

Porównywanie obiektów

- operator porównania ==
 - stwierdza, czy referencje obu argumentów są identyczne
- metoda *equals*
 - na poziomie *Object* jest tożsama z “==”
 - przedefiniowana może opierać się na porównywaniu zawartości obiektów
 - np. *equals* z klasy *String* porównuje, czy łańcuchy obu obiektów są tej samej długości i składają się ze znaków o takich samych kodach
 - nowo tworzone obiekty, jeśli będą ze sobą porównywane powinny przedefiniować metodę *equals* (oraz *hashCode*)

Object

- *hashCode* – zwraca wartość kodu mieszającego dla tego obiektu
 - mapuje obiekt na jednolity skalar zazwyczaj typu *int*
 - wartość ta musi być taka sama przy każdym wywołaniu tej metody na pojedynczym obiekcie
 - jeśli dwa obiekty są równe według metody *equals()* to oba muszą mieć takie same wartości kodu mieszającego
 - nie jest wymagane, aby dla dwóch obiektów, które są różne według metody *equals()*, wartości kodów również były różne, aczkolwiek różne kody dla różnych obiektów mogą poprawić wydajność kodu
 - jest używana przez mechanizmy wbudowane przez niektóre kolekcje np. *Hashtable*, *HashSet*, *HashMap*

Object

- `equals(Object obj)` – zwraca prawdę, jeśli `obj` jest równy obiektowy na rzecz którego metoda jest wywoływana
 - dla każdego nie pustego obiektu `x`, `x.equals(x)` powinno zwracać `true`
 - dla wszystkich nie pustych `x` i `y`, `x.equals(y)` zwraca `true`, jeśli `y.equals(x)` zwraca `true`
 - dla wszystkich nie pustych `x`, `y` i `z`, jeśli `x.equals(z)` zwraca `true` i `y.equals(z)` zwraca `true`, to `x.equals(y)` musi zwracać `true`
 - dla każdego nie pustego obiektu `x`, `x.equals(null)` zawsze zwraca `false`

Przeciążanie metod

- Przeciążanie (*overload*) metody oznacza zdefiniowanie nowej metody o tej samej nazwie, co metoda już istniejąca w klasie, ale z inną listą argumentów
 - różnica tylko w zwracanym typie wartości i modyfikatorze nie jest wystarczająca do utworzenia metody przeciążonej, ponieważ kompilator musi rozpoznać, które ciało metody wstawić pod wywoływany kod
- Przykłady:

```
public void println(String s) { /* ... */ }
public void println(double d) { /* ... */ }
public void println(int i) { /* ... */ }
```

Przeciążanie konstruktorów

- W klasie można zdefiniować kilka konstruktorów różniących się rodzajem lub liczbą argumentów
- Przykłady:

```
public Employee() { /* ... */ }
public Employee(String name) { /* ... */ }
public Employee(String name, int salary) { /* ... */ }
```

- Aby nie powtarzać kodu konstruktor może poprzez referencję *this* wywołać inny konstruktor z tej samej klasy
 - wywołanie innego konstruktora musi nastąpić w pierwszej znaczącej linii konstruktora

Przeciążanie konstruktorów

```
public class Employee {  
    private String name;  
    private int salary;  
  
    public Employee() {  
        this("VIP");  
    }  
  
    public Employee(String name) {  
        this(name, 0);  
    }  
  
    public Employee(String name, int salary) {  
        this.name = name;  
        this.salary = salary;  
    }  
}
```

Pytania sprawdzające

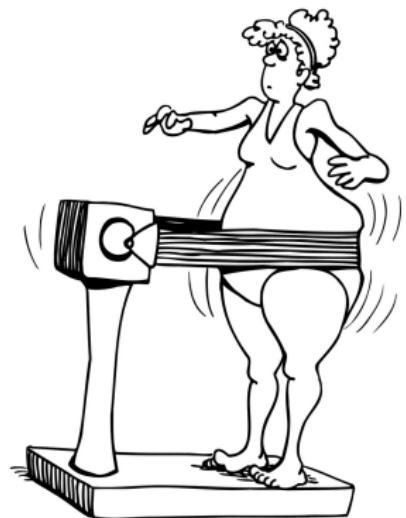
- Na czym polega podejście obiektowe w programowaniu?
- Jaka jest relacja pomiędzy obiektem a klasą?
- Na czym polega abstrakcja danych?
- Jakie zalety ma hermetyzacja klasy?
- Opisz relację asocjacji/agregacji/kompozycji
- Na czym polega relacja dziedziczenia?
- Jak dziedziczenie jest realizowane w Javie?
- Czy klasa potomna dziedziczy konstruktory?
- Omów modyfikatory dostępu

Pytania sprawdzające

- Jak można przeddefiniować metodę w klasie potomnej?
- Na czym polega polimorfizm?
- Czy poprzez referencję typu bazowego można wywołać metodę charakterystyczną dla klasy potomnej?
- Za pomocą jakiego operatora można sprawdzić, czy referencja jest określonego typu?
- Jak przebiega utworzenie nowego obiektu i jego inicjalizacja?
- Podaj przykłady metod dziedziczonych z klasy *Object*?
- W jaki sposób sprawdzić, czy dwa różne obiekty są identyczne?
- Jakie warunki są konieczne do przeciążenia metody?

Ćwiczenia

- Ćwiczenie 5.1:
inicjacja hierarchii obiektów
- Ćwiczenie 5.2:
*zbudowanie hierarchii obiektów;
wykorzystanie dziedziczenia
i polimorfizmu*



Elementy: *static*

- Elementy klasy oznaczone jako *static* przynależą do klasy i są wspólne dla wszystkich obiektów utworzonych z tej klasy
- Ponieważ są unikatowe dla całej rodziny obiektów danego typu
 - można w nich przechowywać dane, wspólne dla całej rodziny
 - kontrolować liczbę instancji obiektów danego typu
 - udostępniać metody, które nie muszą operować na danych obiektu
- Do elementów statycznych można się odwoływać bez konieczności tworzenia obiektu – poprzez podanie nazwy klasy

Elementy: static

```
public class Test {  
    static int a;  
    int b;  
    public static void go() { /* ... */ }  
    public void show() { /* ... */ }  
}
```

```
// elementy statyczne przez nazwe klasy  
Test.a = 10;  
Test.go();
```

```
// elementy obiektu tylko przez referencje  
Test test = new Test();  
test.b = 10;  
test.show();
```

Blok static

- Blok statyczny jest wykonywany przed utworzeniem jakiegokolwiek obiektu danego typu
- Jest stosowany do przeprowadzenia inicjacji środowiska niezbędnego do funkcjonowania obiektu
- W bloku statycznym nie można korzystać z obiektów typu, w którym jest osadzony

```
public class Database {  
    static {  
        // załaduj sterownik  
        new sun.jdbc.odbc.JdbcOdbcDriver();  
    }  
    // ...  
}
```

Wzorzec singleton

- Wzorzec projektowy gwarantujący powstanie tylko jednej instancji obiektu danego typu
- Często stosowany dla obiektów, które nie powinny występować w aplikacji w więcej niż jednym egzemplarzu
- Wykorzystuje pole statyczne do zapamiętania referencji do obiektu oraz ogranicza tworzenie obiektów przez obecność prywatnego konstruktora
- Przykłady:
 - *Company*
 - *ConnectionPool*
 - *SecurityManager*

Wzorzec singleton

```
public class Company {  
    // zmienna trzymajaca referencje  
    private static Company instance;  
  
    // prywatny konstruktor  
    private Company() { /* ... */ }  
  
    // metoda zwracajaca referencje  
    public static Company getInstance() {  
        if (instance == null) {  
            instance = new Company();  
        }  
        return instance;  
    }  
  
    // inne, obiektowe składniki klasy  
}
```

Elementy: *final*

- Słowo kluczowe *final* może być zastosowane do:
 - klasy – klasa nie może być specjalizowana (być podstawą do dziedziczenia)
 - metody – metoda nie może zostać przeddefiniowana
 - zmiennej – nie można zmienić wartości takiej zmiennej
- *final* stosuje się, aby zagwarantować niezmiennosć zachowania obiektu lub metody (aby wykluczyć polimorfizm)

Elementy: *final*

- *final* zastosowane dla zmiennych tworzy stałe:

```
public final int TIME_LIMIT = 1000;  
public final String GREETING = "Hello";
```

- Wartość zmiennej oznaczonej jako *final* i *static* musi być nadana w trakcie inicjacji lub w bloku statycznym w danej klasie
- Wartość zmiennej oznaczonej jako *final* musi być nadana w trakcie inicjacji lub do końca każdego z konstruktorów

Klasa abstrakcyjna

- Klasa abstrakcyjna umożliwia stworzenie klasy bazowej, w której nie wszystkie metody będą zdefiniowane (zaimplementowane)
- Tworzymy klasę abstrakcyjną, jeśli:
 - wszystkie obiekty z danej rodziny powinny posiadać określone zachowanie (metody), ale na poziomie klasy bazowej nie można podać rozsądnej implementacji
 - chcemy zablokować możliwość tworzenia obiektów typu bazowego
 - chcemy tworzyć szablonowe metody oparte o wyniki zwracane przez metody klas konkretnych
- Nagłówek klasy abstrakcyjnej lub niezdefiniowanej metody musi zawierać słowo *abstract*

Klasa abstrakcyjna

```
public abstract class Vehicle {  
    public abstract double getDistance();  
    public abstract double getFuelUsage();  
    public double getEfficiency() {  
        return getFuelUsage() / getDistance();  
    }  
}
```

- W powyższym przykładzie:

- metody *getDistance* i *getFuelUsage* nie mogą być zdefiniowane dla abstrakcyjnego pojęcia, jakim jest *Vehicle*
- zostaną zdefiniowane w konkretnych implementacjach klas np. *Plane*, *Car*, *Ship*
- metoda licząca efektywność danego pojazdu może być już zdefiniowana na podstawie istniejących metod

Klasa abstrakcyjna

- Klasą abstrakcyjną musi być klasa, która ma co najmniej jedną metodę abstrakcyjną
- Klasa dziedzicząca (potomna) po klasie abstrakcyjnej musi zdefiniować wszystkie metody abstrakcyjne
 - w przeciwnym razie musi być zadeklarowana jako abstrakcyjna
- Klasa abstrakcyjna może posiadać konstruktor
 - nie można powołać obiektu klasy abstrakcyjnej, natomiast konstruktor jest wywoływany przez klasy potomne w czasie tworzenia obiektów
- Klasa abstrakcyjna może posiadać elementy statyczne oraz blok statyczny

Interfejs

- **Interfejs** to lista zadeklarowanych metod – bez ich definicji
 - interfejs jest podobny do klasy abstrakcyjnej, w której wszystkie metody są abstrakcyjne
 - interfejs nie może zawierać deklaracji atrybutów – chyba, że są oznaczone jako *final*
 - wszystkie metody, nawet jeśli tego nie zadeklarujemy, są publiczne

Interfejs

- Jest sposobem na zapewnienie dziedziczenia wielobazowego
 - klasa może implementować zachowanie zadeklarowane w kilku interfejsach
 - interfejs może dziedziczyć (rozszerzać) dowolną liczbę innych interfejsów
 - klasa implementująca dany interfejs jest instancją typu reprezentowanego przez niego
 - klasa implementująca interfejs zobowiązuje się do zdefiniowania wszystkich metod w nim zadeklarowanych; w przeciwnym razie musi być oznaczona jako klasa abstrakcyjna

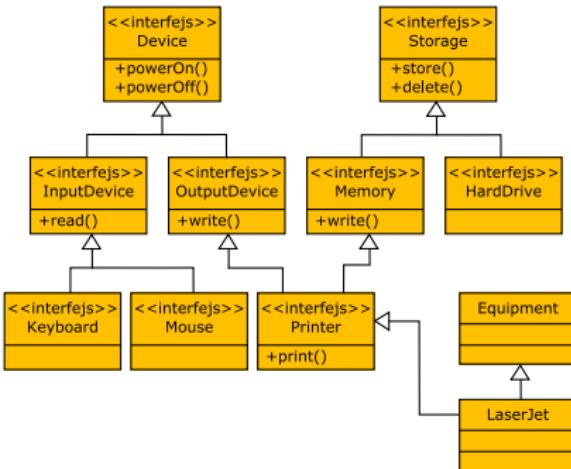
Interfejs – przykład

```
public interface Printable {  
    public void printInfo();  
}
```

```
public class Report implements Printable {  
    /* metody klasy */  
    public void generateReport() {  
        //...  
    }  
    /* definicja metody z interfejsu */  
    public void printInfo() {  
        // ...  
    }  
}
```

Interfejs

```
public interface Printer  
    extends Memory,  
           OutputDevice {  
  
    public void print();  
}  
  
public class LaserJet  
    extends Equipment  
    implements Printer {  
    // ...  
}
```



- Interfejs może dziedziczyć z jednego lub kilku interfejsów

Interfejsy

- Pozwalają na deklarowanie zachowania klasy, które pochodzi z kilku niezależnych dziedzin biznesowych
 - Amfibia – jest samochodem, ale może też pływać
 - Ptak – jest zwierzęciem, ale też obiektem latającym
 - Człowiek – jest pracownikiem, ale też zwierzęciem
- Interfejs umożliwia uchwycenie podobieństw między obiektami bez sztywnej struktury hierarchii dziedziczenia
- Interfejsy pozwalają na modelowanie zachowania obiektów bez odkrywania szczegółów implementacji ich modelu danych
- Interfejsy mogą też stanowić rolę znacznikową dla obiektów, na których będą wykonywane specyficzne operacje
 - *Cloneable, Serializable, Remote*

Klasy wewnętrzne

- Klasa wewnętrzna jest definiowana jako składnik innej klasy
- Ma ona dostęp do wszystkich składników klasy zewnętrznej (również prywatnych)
 - potrzeba dostępu do elementów prywatnych z poziomu innego obiektu jest częstym powodem tworzenia klas wewnętrznych
 - sama klasa wewnętrzna może być prywatna i niedostępna na zewnątrz
- Jej instancja jest powoływana w kontekście instancji klasy zewnętrznej

Klasy wewnętrzne

```
public class Outer {  
    private int a;  
    public void callInner() {  
        Inner i = new Inner();  
        i.assign();  
    }  
    class Inner {  
        public void assign() {  
            a = 10;  
        }  
    }  
}
```

```
// utworzenie obiektu klasy wewnętrznej  
Outer outer = new Outer();  
Inner inner = outer.new Inner();  
inner.assign();
```

Jakość modelu obiektowego

• Zwartość/Spójność (*cohesion*)

- miara, jak bardzo klasa lub grupa klas przyczynia się do realizacji określonego celu
- należy dążyć do jak największej zwartości
 - klasy (lub grupy klas) powinny być skupione na realizacji określonego celu – zazwyczaj jednego
 - klasa, która realizuje kilka różnych zadań ma słabą zwartość

Jakość modelu obiektowego

● Sprzężenie (*coupling*)

- miara, jak dwie lub więcej klas jest od siebie wzajemnie zależnych
- należy dążyć do jak najmniejszego sprzężenia między obiekta mi lub grupami obiektów
 - obiekty powinny być od siebie zależne tylko w minimalnym zakresie niezbędnym do realizacji określonych zdań
 - jeśli zależność (liczne połączenia) między klasami jest duża, trudno o dobre ponowne wykorzystanie kodu

Dziedziczenie czy delegacja

● Dziedziczenie

- jest relacją, która zachodzi między obiektyami uogólnionym i specyficzny (generalizacja-specjalizacja)
- powinna być wybierana tylko, jeśli klasa potomna rozwija (specjalizuje) możliwości klasy bazowej, ale cel istnienia danego typu nie ulega zmianie

Dziedziczenie czy delegacja

- Delegacja
 - oznacza delegowanie części zadań, które nie stanowią głównego celu istnienia typu, innemu obiekowi
 - realizacja zadań pobocznych częściej ulega modyfikacjom i dzięki wydelegowaniu ich realizacji do innego obiektu, kod staje się łatwiejszy w utrzymaniu
- Przykłady:
 - Klasa *Group*, zarządzająca użytkownikami nie jest specjalizacją kolekcji danych *List*; samo kolekcjonowanie użytkowników deleguje do innego obiektu
 - jej główne zadania związane są z operacjami na użytkownikach (sprawdzanie uprawnień, zmiana danych)
 - Klasa *OrderedList* jest rozwinięciem kolekcji danych *List*; jej celem istnienia jest kolekcjonowanie danych

Podsumowanie

- Modelowanie obiektowe
- Podstawowe pojęcia obiektowe
 - Relacje między obiektami
 - Modyfikatory, elementy *static* i *final*
- Klasy i interfejsy
 - Dziedziczenie, polimorfizm,
 - Klasy abstrakcyjne, interfejsy
 - Klasy wewnętrzne

Pytania sprawdzające

- Jakie cechy mają elementy statyczne klasy?
- W jaki sposób można zaimplementować wzorzec singletona?
- Jakie konsekwencje pociąga za sobą użycie słowa kluczowego *final* w stosunku do klasy/metody/atrributu?
- Kiedy należy nadać wartość początkową zmiennej finalnej?
- Jak utworzyć klasę abstrakcyjną i jakie ma ona właściwości?
- Jakie jest zastosowanie klas abstrakcyjnych?
- Jaka jest struktura zawartości interfejsu?
- Do czego mogą być przydatne interfejsy?
- Jakie właściwości mają klasy wewnętrzne?
- Jakich miar użyjesz w badaniu jakości modelu obiektowego?

Ćwiczenia

- Ćwiczenie 5.3:
zastosowanie wzorca Singleton
- Ćwiczenie 5.4:
*zbudowanie hierarchii obiektów;
klasa abstrakcyjna, interfejsy, wywołania
polimorficzne, rzutowanie typów*



Plan szkolenia

- 1 Wprowadzenie do technologii Java
- 2 Proste obiekty i typy danych
- 3 Zasady budowania aplikacji w Javie
- 4 Podstawy składni języka
- 5 Koncepcje programowania obiektowego
- 6 Obsługa błędów i wyjątków**
- 7 Środowisko systemowe i operacje we/wy
- 8 Najczęściej popełniane błędy w Javie

Plan modułu

6

Obsługa błędów i wyjątków

- rodzaje sytuacji wyjątkowych
- reguły obsługi wyjątków
- tworzenie własnych typów wyjątków

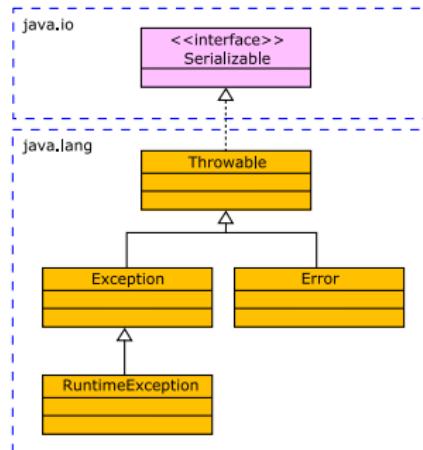
Czym są sytuacje wyjątkowe

- W realizowanych programach można wyróżnić trzy rodzaje sytuacji wyjątkowych:
 - błędy wynikające ze środowiska uruchomieniowego
 - brak dostępnej pamięci operacyjnej
 - błąd biblioteki systemowej lub *JVM*
 - błędy powodowane przez otoczenie aplikacji
 - niedostępność urządzeń wejścia/wyjścia
 - przerwane połączenia sieciowe
 - błędy spowodowane przez programistę
 - korzystanie z niezainicjowanych zmiennych
 - przekroczenie indeksów tablicy
 - dzielenie przez zero

Obsługa sytuacji wyjątkowych

- Każdy błądwyławany przez JVM jest opisany przez odpowiedni obiekt pochodzący od typu *Throwable*
 - błędy wynikające ze środowiska uruchomieniowego
 - błędy pochodzące z rodziny *Error*
 - nie muszą i wręcz nie powinny być obsługiwane przez programistę
 - sugerowane jest przekazanie informacji użytkownikowi o wystąpieniu błędu
 - błędy powodowane przez otoczenie aplikacji
 - błędy pochodzą z rodziny *Exception*
 - muszą być obowiązkowo obsługiwane;
 - programista powinien dostarczyć kod uruchamiany w przypadku wystąpienia błędu
 - błędy spowodowane przez programistę
 - błędy pochodzą z rodziny *RuntimeException*
 - nie muszą być obsługiwane; idealnie napisany program ich nie posiada
 - są dobrym źródłem informacji przy usprawnianiu aplikacji

Obsługa sytuacji wyjątkowych



- Wyjątki wywodzące się z klas *Error* oraz *RuntimeException* to wyjątki **niekontrolowane**
- Wyjątki **kontrolowane** pochodzą z klasy *Exception* z pominięciem wyjątków należących do klasy *RuntimeException*

Blok chroniony

- Kod, który może wygenerować wyjątek powinien (lub musi) być uruchamiany w bloku chronionym
- Po bloku chronionym umieszcza się bloki obsługi wyjątków wykonywane w przypadku wystąpienia konkretnej sytuacji krytycznej
- Z blokiem chronionym można związać dowolnie dużo bloków obsługi wyjątków

```
// blok chroniony
try {
    // operacje, które potencjalnie powodują wyjątki
} catch (IOException e) {
    // kod obsługi wyjątku
} catch (AWTException e) {
    // kod obsługi wyjątku
}
```

Sterowanie operacjami

```
public void method() {  
    oper1();  
    try {  
        oper2();  
        oper3();  
    } catch(Exception e) {  
        // obsługa wyjątku  
        oper4();  
    }  
    oper5();  
}
```

- Jeśli wyjątek typu *Exception* wystąpi w trakcie realizacji *oper2*, wówczas sterowanie jest przekazane do bloku *catch* i wykonana jest *oper4*, a następnie *oper5*
- Jeśli nie wystąpi żaden wyjątek w bloku *try*, to po realizacji *oper3* wykonana zostanie *oper5*

Sterowanie operacjami

```
public void method() {  
    oper1();  
    try {  
        oper2();  
        oper3();  
    } catch(Exception e) {  
        // obsługa wyjątku  
        oper4();  
    }  
    oper5();  
}
```

- Jeśli podczas realizacji *oper2* lub *oper3* wystąpi wyjątek spoza rodziny *Exception*, wówczas przerwane zostanie wykonywanie metody i sterowanie zostanie przekazane do metody wywołującej
 - nie zostanie wykonana *oper4* ani *oper5*

Kolejność przechwytywania

- Instrukcje w bloku chronionym mogą generować różne wyjątki – obsługa zostanie przekazana do najbliższego bloku *catch* obejmującego powstały wyjątek
- Umieszczając bloki *catch* należy pamiętać, aby wyjątki bardziej szczegółowe umieszczać bliżej *try*

```
try {
    f.readLine();
} catch(FileNotFoundException e) {
    // obsługa
} catch (IOException e) {
    // obsługa
} catch (AccessDeniedException e) {
    // obsługa
}
```

- Nie należy dopuszczać do sytuacji przedstawionej powyżej – w Javie 5.0+ kompilator wygeneruje błąd nieosiągalnego kodu

Przechwytywanie wielu wyjątków

- Zdarza się, że w bloku chronionym przechwytyujemy wiele wyjątków, a sposób reakcji na sytuacje wyjątkowe jest identyczny
- W takim przypadku mamy wiele sekcji *catch* z wielokrotnie zduplikowanym kodem
- Od Java SE 7 można z jedną sekcją *catch* związać wiele wyjątków, oddzielając je znakiem |

```
try { /* ... */ }
catch (IOException|SQLException ex) {
    ex.printStackTrace();
}
```

- parametr sekcji *catch* jest niejawnie finalny

Obsługiwać/przekazywać

- Miejsce powstania wyjątku nie zawsze jest odpowiednie do stworzenia rozsądnego mechanizmu jego obsługi
- Jeśli nie można zapewnić dobrej obsługi wyjątku, to można go przekazać do metody nadzędnej
 - wówczas metoda nadzędna może obsłużyć wyjątek lub przekazać go dalej
- Metoda nie obsługująca wyjątku, a wykonująca operacje, które muszą być chronione musi zadeklarować listę wyjątków, jakie potencjalnie może powodować jej wywołanie
- Wywołanie metody, która ma zadeklarowaną listę potencjalnie generowanych wyjątków musi odbywać się w bloku chronionym
- Metoda może zadeklarować dowolnie długą listę wyjątków

Obsługiwać/przekazywać

```
public void met1() {  
    try {  
        met2();  
    } catch (MyException e) {  
        // obsługa  
    }  
}  
  
void met2() throws MyException {  
    oper1();  
    oper2(); // powoduje wyjątek  
}
```

Przeddefiniowywanie metod

- Metoda, która ma zadeklarowaną listę wyjątków może być przeddefiniowana pod warunkami:
 - nowa metoda nie będzie rozszerzać listy wyjątków kontrolowanych
 - nowa metoda nie będzie potencjalnie generować wyjątków bardziej ogólnych od tych w metodzie bazowej
- Ze względu na zachowanie polimorficzne, klient metody bazowej nie może być "zaskoczony" nagłym pojawiением się większej liczby wyjątków lub szerszej rodziny wyjątków
 - podobnie jak węższym zakresem dostępności
- Metoda przeddefiniująca może zmniejszyć listę wyjątków lub je całkowicie zlikwidować
- Metoda przeddefiniująca może rozszerzyć listę wyjątków o wyjątki niekontrolowane

Blok *finally*

- W sytuacji, kiedy chcemy zagwarantować wykonanie pewnego bloku instrukcji bez względu na powstały wyjątek, to umieszczamy je w bloku *finally*
- Blok *finally* jest wykonywany zawsze bez względu na to:
 - czy blok *catch* istnieje
 - czy blok *catch* obsłużył powstały wyjątek
 - czy został wygenerowany wyjątek

Blok finally

- Blok *finally* jest wykonywany nawet, jeśli w bloku *try* znajduje się instrukcja *return*

```
try {
    openConnection();
    // wykonaj zapytania
    // odczytaj wyniki
    return;
} catch (ConnectionException e) {
    // obsługa
} finally {
    closeConnection();
}
```

Własne wyjątki

- Własne wyjątki można tworzyć bazując na jednej z klas rodziny *Exception*
- Wyjątek może zawierać dodatkową informację tekstową
- Wyrzucenie wyjątku odbywa się komendą **throw**
- Wyjątek (obiekt) powinien być utworzony w chwili wyrzucenia
 - zawiera m.in. informacje o stosie wywoływanych metod

Własne wyjątki – przykład

```
public class MyException extends Exception {  
    public MyException() {}  
    public MyException(String s) { super(s); }  
}
```

```
void met1() throws MyException {  
    // ...  
    if (wrong) throw new MyException();  
    // ...  
}
```

Kilka dobrych rad

- Jeśli jesteśmy zmuszeni do zastosowania bloku chronionego, w bloku *catch* nie wolno ignorować obsługi wyjątku
 - co najmniej należy odnotować powstanie wyjątku
 - wygodne jest użycie metody z klasy *Exception*: *printStackTrace()*
- Zignorowana obsługa wyjątku, na zawsze pomija milczeniem błąd, który wcześniej czy później da o sobie znać, ale znaleźć go będzie trudno
- Wyjątki należy stosować nie tylko w sytuacjach błędów aplikacji
 - każda sytuacja wyjątkowa z punktu widzenia logiki aplikacji może być obsługiwana przez wyjątek
- Wyjątki przerzucane z metod niskopoziomowych można opakowywać w inne klasy wyjątków
 - opakowanie wyjątków gwarantuje przechowanie informacji o stosie, która może być przydatna do odnalezienia błędu

Podsumowanie

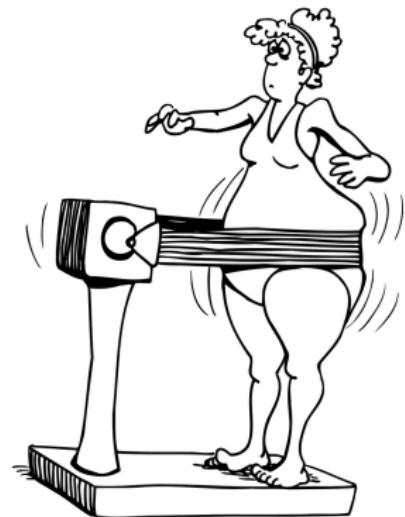
- Rodzaje sytuacji wyjątkowych
- Reguły obsługi wyjątków
- Tworzenie własnych typów wyjątków

Pytania sprawdzające

- Jakie są kategorie sytuacji wyjątkowych i które z nich należy obsłużyć programowo?
- Jaka jest podstawowa hierarchia dziedziczenia klas wyjątków?
- Które z wyjątków są niekontrolowane?
- Jaką składnię ma blok chroniony?
- Co to jest stos wywołań?
- Kiedy obsługę wyjątku należy przekazać do metody nadzędnej?
- Jak można prześlonić metodę deklarującą możliwość wyrzucania wyjątków?
- Jak tworzyć własne klasy wyjątków i jak je wyrzucać?

Ćwiczenia

- Ćwiczenie 6.1:
*utworzenie własnej klasy wyjątku;
powstanie i przechwycenie sytuacji
wyjątkowej*



Plan szkolenia

- 1 Wprowadzenie do technologii Java
- 2 Proste obiekty i typy danych
- 3 Zasady budowania aplikacji w Javie
- 4 Podstawy składni języka
- 5 Koncepcje programowania obiektowego
- 6 Obsługa błędów i wyjątków
- 7 Środowisko systemowe i operacje we/wy
- 8 Najczęściej popełniane błędy w Javie

Plan modułu

7

Środowisko systemowe i operacje we/wy

- korzystanie z klas systemowych
- kolekcje danych
- typy generyczne w kolekcjach
- klasy opakowujące
- adnotacje
- obsługa operacji wejścia-wyjścia

Sterowanie aplikacją

- Podczas uruchamiania aplikacji można przekazać jej parametry z linii poleceń:

```
java Main ala kot "duzy pies"
```

- Parametry są dostępne jako argument metody *main*:

```
public class Main {  
    public static void main(String[] args) {  
        for (int i=0; i<args.length; i++) {  
            System.out.println(args[i]);  
        }  
    }  
}
```

- Badając długość tablicy można stwierdzić, czy zostały przekazane parametry do aplikacji

Zmienne systemowe

- Środowisko, w którym działa aplikacja dostarcza informacji poprzez zmienne systemowe

```
Properties p = System.getProperties();
p.list(System.out);
```

- Zmienne gromadzone w kolekcji *Properties* są kojarzone z nazwą-kluczem

```
String os = p.getProperty("os.name");
```

Zmienne systemowe

- Obiektu *Properties* można używać do przechowywania własnych danych. Można je zapisywać i odtwarzać poprzez metody *store* i *load*
- Do środowiska systemowego można wprowadzić własne zmienne w linii wywołania programu

```
java -Dmoja.zmienna=wartosc SystemVariables
```

Dystrybucja aplikacji

- Aplikacja zawiera zazwyczaj dosyć pokaźną liczbę klas i innych plików towarzyszących
- Z plików składających się na aplikację można utworzyć archiwum, które jest łatwiejsze w dystrybucji

```
jar cf <nazwa_archiwum> <nazwa_katalogu_aplikacji>
```

Dystrybucja aplikacji

- Z odpowiednio przygotowanego archiwum można bezpośrednio uruchamiać aplikację
 - należy przygotować plik MANIFEST.MF, który będzie zawierał informację o klasie zawierającej metodę *main*:

```
Main-Class: mod6.SystemVariables
```

- przygotować archiwum jar:

```
jar cfm <nazwa archiwum> manifest.mf <pliki_class>
```

- uruchomić aplikację:

```
java -jar <nazwa_archiwum>
```

Klasa StringBuffer

- Klasa *String* reprezentuje obiekt, który nie może zmieniać swojej zawartości
- Wszelakie operacje wykonywane na łańcuchach znaków w oparciu o *String* oznaczają generowanie nowych obiektów
- Do budowania złożonych łańcuchów znaków lepiej jest używać klasy *StringBuffer*
- Obiekt *StringBuffer* reprezentuje rozszerzalny bufor znaków, do którego można dodawać kolejne poprzez przeciążoną metodę *append*:

```
StringBuffer sb = new StringBuffer();
sb.append("ala").append("ma").append("kota");
String s = sb.toString();
```

Klasa StringBuffer

- *StringBuffer* pozwala na dodatkowe operacje na buforze:
 - zamianę znaków
 - wprowadzanie znaków między już istniejące
 - wycinanie fragmentów łańcucha

Klasa `StringBuilder`

- Klasa o podobnym zastosowaniu jak `StringBuffer`, lecz zalecana jako szybsza w większości implementacji
- W przeciwieństwie do `StringBuffer` nie zapewnia synchronizacji i bezpieczeństwa w programach wielowątkowych
- `StringBuilder` i `StringBuffer` posiadają takie same metody i konstruktory

Klasa Locale

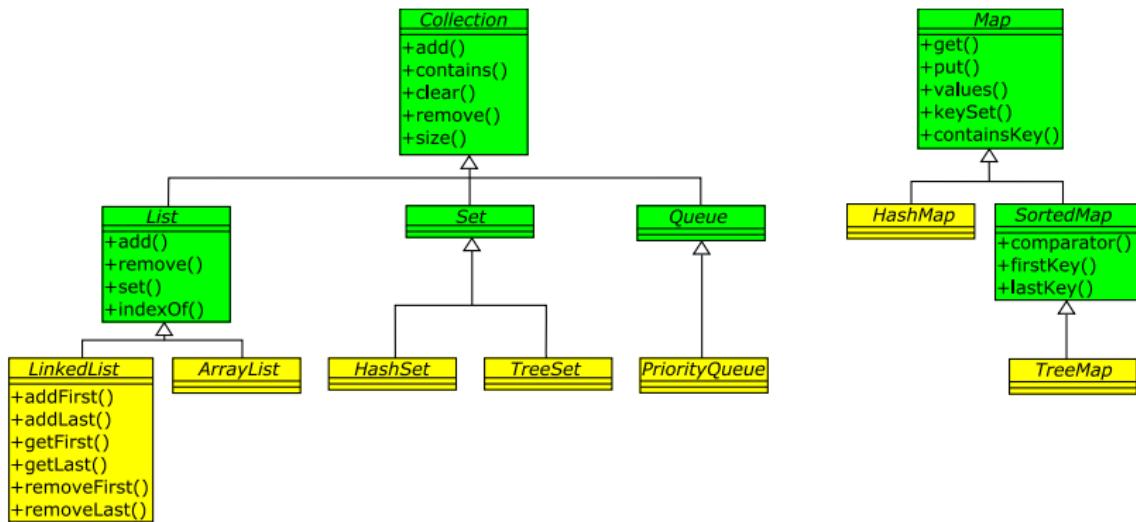
- Tworzy lokalizator zawierający informacje o aktualnej lokalizacji, takie jak: język, kraj oraz opcjonalny wariant
- Lokalizator sam w sobie posiada niewielką funkcjonalność, lecz jest często wykorzystywany przez klasy prezentujące dane użytkownikowi
- Dostarcza innym klasom informacje o sposobie prezentacji danych wyjściowych
- Przykład wykorzystania:

```
Locale polska = new Locale("pl", "PL");
NumberFormat nf = NumberFormat.getCurrencyInstance(polska);
double pensja = 1233.12;
System.out.println(nf.format(pensja));
```

Collection API

- Kolekcja reprezentuje zbiór obiektów zwanych jej elementami
- Kolekcja może zapewniać:
 - unikatowość elementu w zbiorze – *Set*
 - uporządkowanie elementów – *List*
 - powiązanie elementu z kluczem – *Map*
- Kolekcja daje jednolity interfejs pracy z kontenerami danych
- Kolekcja umożliwia spójny sposób przeglądania zbioru
- Można zdefiniować typ elementów wewnętrz kolekcji

Kolekcje



Kolekcje – interfejs Set

- Java SE dostarcza różne implementacje interfejsu *Set*
 - *HashSet* – zbiór nieuporządkowany o unikalnych elementach
 - *TreeSet* – zbiór uporządkowany o unikalnych elementach
 - *LinkedHashSet* – zbiór pamiętający kolejność umieszczonych elementów

Kolekcje – interfejs List

- Java SE dostarcza różne implementacje interfejsu *List*
 - ArrayList* – lista oparta o tablicę, szybki dostęp do elementów, ale wysoki koszt zmian
 - LinkedList* – lista oparta o struktury dynamiczne, niewielki koszt zmian (dodawanie, usuwanie elementów), ale wolniejszy dostęp do elementów; może zachowywać się jak kolejka FIFO

Kolekcje – interfejs Map

- Java SE dostarcza różne implementacje interfejsu *Map*
 - *HashMap* – mapa nieuporządkowana
 - *TreeMap* – mapa uporządkowana
 - *LinkedHashMap* – mapa zapamiętująca kolejność umieszczania elementów

Kontrola typów dla kolekcji (*Generics*)

- Informacja dla kompilatora o typie elementu wewnątrz kolekcji
- Kontrola typów dla kolekcji przeprowadzana w trakcie kompilacji
- Wyeliminowanie rzutowania
- Przykład:

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(new Integer(5));  
list.add(3); // konstrukcja dostępna od Java 5.0
```

- Typ generyczny musi być typem obiektowym

Tworzenie kolekcji

```
Collection<Object> set = new HashSet<Object>();
set.add(new String("ala"));
set.add(new Vehicle());
// ...
set = new ArrayList<Object>();
set.add(new String("ala"));
set.add("jola");
set.add(new Vehicle());

Map<String, Object> map = new HashMap<String, Object>();
map.put("ala", "kot");
map.put("pojazd", new Vehicle());
// ...
Vehicle v = (Vehicle) map.get("pojazd");
Collection values = map.values();
Set keys = map.keySet();
```

Typy generyczne

- Od Javy SE 7 można pominąć typy generyczne (pozostawiając nawiasy ostre – “diamenty”) wszędzie tam, gdzie kompilator będzie w stanie ich się domyślić z kontekstu

```
Map<String, List<String>> myMap = new HashMap<>();
```

Przeglądanie kolekcji

- Kolekcje zwracają *Iterator*, dzięki któremu można sekwencyjnie przeglądać elementy kolekcji

```
Set<Object> set = new HashSet<Object>();
for (Iterator i = set.iterator(); i.hasNext();) {
    Object element = i.next();
    if (shouldBeRemoved(element)) {
        i.remove(); // poprzez iterator
    }
}
```

- Lista zwraca specjalizowany *ListIterator* umożliwiający przeglądanie dwukierunkowe
 - *ListIterator* posiada m.in. metody: *hasPrevious()*, *previous()*

Przeglądanie kolekcji

- Od Javy SE 5.0 kolekcje można przeglądać za pomocą nowej rozszerzonej pętli *for* ("for-each")

```
Collection<String> set = new ArrayList<String>();
set.add("ala");
set.add("kot");
set.add("pies");
for (String s : set) {
    System.out.println(s);
}
```

Klasy opakowujące

- W dystrybucjach środowiska Java starszych niż wersja 5.0, biblioteki oparte o obiekty (*Collection API*) były niedostępne dla typów prostych
 - aby umieścić typ prosty w kolekcji należało go umieścić w jego typie opakowującym (*int – Integer, float – Float, itd.*)
- Typy opakowujące nie pozwalają na zmianę raz zamkniętej w nich wartości – służą tylko jako opakowanie
- Wraz z Javą 5.0 pojawił się mechanizm **autoboxingu**, dzięki czemu niepotrzebne stało się ręczne opakowywanie typów prostych w celu umieszczenia ich w kolekcji
- Jednakże typy proste nie mogą być używane jako typy generyczne i należy w tym miejscu używać typów opakowujących

Klasy opakowujące

- Elementy można umieszczać w kolekcji posługując się automatycznym mechanizmem tzw. autoboxingiem lub można dokonywać ręcznej konwersji na typ opakowujący

```
Collection<Integer> set = new ArrayList<Integer>();
Collection<int> set2 = new ArrayList<int>(); // zle!!!
set.add(30);
set.add(10);
set.add(new Integer("123"));
set.add(new Integer(1));
int suma = 0;
for (int i : set)
    suma += i;
System.out.println(suma);
```

Adnotacje

- W Javie SE 5.0 wprowadzono mechanizm **adnotacji** umożliwiający zapisywanie metadanych w kodzie źródłowym
- Mogą być alternatywą dla konfiguracji w plikach konfiguracyjnych (np. XML)
- Adnotacje rozpoczynają się znakiem @ np. `@Validate`
- Nie są niezbędnym elementem języka, jednak coraz więcej nowopowstających bibliotek wymaga umiejętności posługiwania się nimi
- Przykład – określenie reguł walidacji pola poprzez adnotację

```
@Validate(length = 50, nullable = false)
public void setName(String name) {
    this.name = name;
}
```

Podstawowe adnotacje Java

• *@Override*

- Oznacza, że dana metoda przedefiniowuje metodę klasy bazowej
- Jeśli metoda o takiej nazwie nie będzie istnieć w klasie bazowej, zostanie wygenerowany błąd

```
@Override
```

```
public String toString() {  
    return super.toString() + " !!!";  
}
```

Podstawowe adnotacje Java

• *@Deprecated*

- Oznacza, że dana metoda jest przeznaczona do usunięcia w następnych wersjach systemu
- Tym samym nie zaleca się jej używania

```
@Deprecated
public void notNeededMethod() {
    // ...
}
```

Podstawowe adnotacje Java

- *@SuppressWarnings*

- Oznacza, że kompilator ma nie generować ostrzeżeń wynikających z konstrukcji metody
- Przyjmuje argument – rodzaj ostrzeżenia, które kompilator ma pominąć

```
@SuppressWarnings({ "deprecation" })
public void useDeprecated() {
    notNeededMethod();
    // ...
}
```

Ćwiczenia

- Ćwiczenie 7.1:

zapoznanie się z przykładami przekazywania argumentów do aplikacji oraz odczytywania zmiennych systemowych

- Ćwiczenie 7.2:

modyfikacja projektu TravelOffice – wprowadzenie i wykorzystanie kolekcji danych

- Ćwiczenie 7.3:

przygotowanie dystrybucji aplikacji



Strumienie

- Mechanizm uszeregowanego przesyłania danych
- Poprzez strumienie można przesyłać dane binarne lub tekstowe pomiędzy dwoma węzłami
 - węzłami mogą być:
 - aplikacje
 - bufor pamięci
 - plik
 - gniazdko sieciowe
 - wątek
- Strumienie są jednokierunkowe – przesyłają dane w jednym kierunku
- Pojedyńczy strumień posiada wejście (input) i wyjście (output)

Strumienie binarne i tekstowe

- Strumienie binarne
 - *InputStream*
 - *OutputStream*
- Służą przesyłaniu danych surowych, które mogą być następnie poddawane interpretacji
- Strumienie tekstowe
 - *Reader*
 - *Writer*
- Służą przesyłaniu danych tekstowych (opartych o *char*)
- Zapewniają odpowiednią konwersję znaków między stronami kodowymi węzła wejściowego i węzła wyjściowego

Strumień wejściowy

- *InputStream* – binarny strumień udostępniający w sposób sekwencyjny dane do odczytu
 - *read()* – zwraca odczytany bajt (w zmiennej typu *int*)
 - *read(byte[] buff)* – wczytuje dane do podanej tablicy
 - *read(byte[] buff, int off, int len)* – wczytuje dane do podanej tablicy, począwszy od *off*, w ilości *len*
 - *markSupported()* – sprawdza, czy strumień może być znacznikowany
 - *mark(int readLimit)* – ustawia znacznik
 - *reset()* – powrót do znacznika
 - *close()* – zamyka strumień

Strumień wyjściowy

- *OutputStream* – binarny strumień pozwalający na zapisywanie danych w sposób sekwencyjny
 - `write(int)` – zapisuje bajt danych do strumienia
 - `write(byte[] buff)` – zapisuje dane z podanej tablicy
 - `write(byte[] buff, int off, int len)` – zapisuje dane z podanej tablicy, począwszy od `off`, w ilości `len`
 - `flush()` – “wypłukuje” dane ze strumienia – zapewnia przesłanie ich do odbiorcy
 - `close()` – zamyka strumień (zapewnia przesłanie ich do odbiorcy)

Strumień wejściowy tekstowy

- *Reader* – sekwencyjny strumień tekstowy udostępniający dane do odczytu
 - *read()* – zwraca odczytany znak
 - *read(char[] buff)* – wczytuje dane do podanej tablicy
 - *read(char[] buff, int off, int len)* – wczytuje dane do podanej tablicy, począwszy od *off*, w ilości *len*
 - *markSupported()* – sprawdza, czy strumień może być znacznikowany
 - *mark(int readAheadLimit)* – ustawia znacznik
 - *reset()* – powrót do znacznika
 - *close()* – zamyka strumień

Strumień wyjściowy tekstowy

- *Writer* – tekstowy strumień pozwalający na zapisywanie danych w sposób sekwencyjny
 - *write(int c)* – zapisuje znak do strumienia
 - *write(char[] buff)* – zapisuje dane z podanej tablicy
 - *write(char[] buff, int off, int len)* – zapisuje dane z podanej tablicy, począwszy od *off*, w ilości *len*
 - *write(String s)* – zapisuje dane z podanego obiektu typu *String*
 - *write(String s, int off, int len)* – zapisuje dane z podanego obiektu *String*, począwszy od *off*, w ilości *len*
 - *flush()* – “wypłukuje” dane ze strumienia – zapewnia przesłanie ich do odbiorcy
 - *close()* – zamyka strumień (zapewnia przesłanie ich do odbiorcy)

Standardowe wejście i wyjście

- W Javie SE mamy do dyspozycji trzy strumienie standardowe:
 - *System.in* – reprezentuje standardowe wejście, w typowym zastosowaniu umożliwia pobieranie znaków z klawiatury
 - *System.out* – reprezentuje standardowe wyjście, w typowym zastosowaniu umożliwia wysyłanie informacji na konsolę
 - *System.err* – reprezentuje standardowe wyjście błędów

Korzystanie z plików

- Współpraca z plikami wymaga otwarcia strumienia związanego z plikiem
- W zależności od tego, czy chcemy pracować z plikiem binarnym, czy tekstowym wybieramy odpowiednią parę strumieni
 - FileInputStream, FileOutputStream*
 - FileReader, FileWriter*

```
FileWriter fw = new FileWriter("output.txt");
fw.write("Ala ma kota");
fw.close();
```

- We współpracy z plikami w celu poprawy efektywności stosuje się filtry buforujące:

```
BufferedWriter bw = new BufferedWriter(
    new FileWriter("output.txt"));
```

Operacje na plikach

- Przy korzystaniu z plików wygodnie jest posługiwać się typem `File`. Pozwala dotrzeć do informacji o pliku, ale nie pozwala z niego odczytywać lub do niego zapisywać. Do zapisu używamy strumieni

```
File file = new File("output.txt");
```

- `canRead, canWrite, exists, getPath` – operacje testujące plik
- `isDirectory, list, mkdir` – operacje na plikach będących katalogami
- `delete, rename` – operacje zarządzania plikami

```
File file = new File("plik.txt");
if (file.exists()) {
    FileReader fr = new FileReader(file);
    // ...
}
```

Obsługa zasobów

- **Zasobami** są obiekty, które muszą zostać zamknięte przed zakończeniem programu (np. pliki)
- Aby zapewnić wykonanie jakiejś sekwencji instrukcji bez względu na to, czy wystąpił wyjątek, czy nie, można wykorzystać konstrukcję *try/finally*

```
static String readFirstLine(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

Obsługa zasobów

- Od Javy SE 7 można to samo osiągnąć prościej za pomocą nowej konstrukcji bloku *try* (tzw. *try with resources*)

```
static String readFirstLine(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(
            new FileReader(path))) {
        return br.readLine();
    }
}
```

- zasoby zadeklarowane w nawiasach za słowem *try* zostaną automatycznie zamknięte
- instrukcja może także posiadać “tradycyjne” sekcje *catch* oraz *finally*
- kod sekcji *catch/finally* jest wykonywany dopiero po zamknięciu zasobów
- zasoby muszą implementować interfejs *AutoCloseable*

Analiza ciągów tekstowych – Tokenizer

- W Javie SE istnieje klasa, która wspiera analizę ciągów tekstowych
- Umożliwia podział tekstu wg zadanego tokenu

```
import java.util.StringTokenizer;

public class TokenizerSample {
    public static void main(String[] args) {
        String aString = "word1 word2 word3";
        StringTokenizer parser = new StringTokenizer(aString);
        while (parser.hasMoreTokens()) {
            String token = parser.nextToken();
            // ...
        }
    }
}
```

Podsumowanie

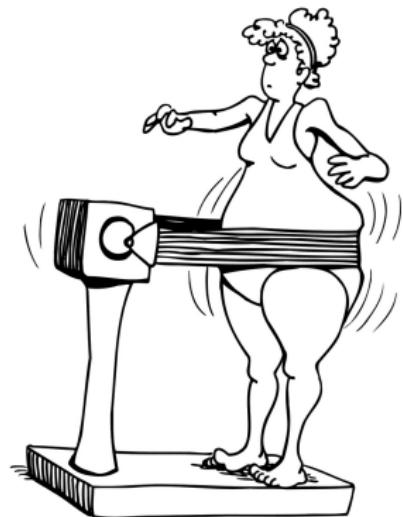
- Korzystanie z klas systemowych
- Kolekcje danych
- Klasy opakowujące
- Obsługa operacji wejścia-wyjścia

Pytania sprawdzające

- Jak można odebrać argumenty przekazywane do programu?
- W jaki sposób można utworzyć dystrybucję aplikacji?
- Jaką rolę w dystrybucji pełni plik manifestu?
- Jakie znasz klasy operujące na łańcuchach tekstowych i jakie są ich główne cechy?
- Scharakteryzuj podstawowe implementacje kolekcji i odwzorowań (jakie interfejsy one implementują)?
- Jakie zastosowanie mają typy generyczne?
- Podaj sposoby przeglądania kolekcji
- Do czego służą klasy opakowujące?
- Podaj przykłady użycia adnotacji
- Wymień klasy bazowe dla strumieni

Ćwiczenia

- Ćwiczenie 7.4:
zapis danych systemu TravelOffice do pliku



Plan szkolenia

- 1 Wprowadzenie do technologii Java
- 2 Proste obiekty i typy danych
- 3 Zasady budowania aplikacji w Javie
- 4 Podstawy składni języka
- 5 Koncepcje programowania obiektowego
- 6 Obsługa błędów i wyjątków
- 7 Środowisko systemowe i operacje we/wy
- 8 Najczęściej popełniane błędy w Javie

Plan modułu

8

Najczęściej popełniane błędy w Javie

- dostęp do elementów instancjnych z metod statycznych
- literówki w nadpisywanych metodach
- porównywanie wartości typów prostych i referencyjnych
- przekazywanie parametrów do metod
- dławienie wyjątków
- indeksowanie elementów tablic
- konstruktor domyślny
- konwencje nazewnicze
- wyjątek NullPointerException

Najczęściej popełniane błędy przez programistów Javy

- Lista błędów, które pojawiają się bardzo często na początku pracy z Java
- Najczęściej one są przyczyną błędów wskazywanych przez kompilator lub środowisko programistyczne
- Ich znajomość potrafi zaoszczędzić wiele godzin poszukiwania przyczyn błędów

Dostęp do niestatycznych pól ze statycznych metod

```
public class Example1 {  
    public String memberVariable = "somedata";  
  
    public static void main(String args[]) {  
        // Proba dostepu do zmiennej niestatycznej  
        System.out.println("Blad komplilacji "  
                           + memberVariable);  
    }  
}
```

Dostęp do niestatycznych pól ze statycznych metod – Rozwiązanie

```
public class Example1Solution {  
    public String memberVariable = "somedata";  
  
    public static void main(String args[]) {  
        // Aby mieć dostęp do pola należy stworzyć obiekt  
        Example1Solution solution  
            = new Example1Solution();  
        System.out.println("Teraz jest OK "  
                           + solution.memberVariable);  
    }  
}
```

Literówka w nazwie przeddefiniowanej metody

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class Example2 extends WindowAdapter {
    // Powinno być windowClosed!
    public void windowClose(WindowEvent e) {
        // Zakończ
        System.exit(0);
    }
}
```

Literówka w nazwie przeddefiniowanej metody – Rozwiązanie

- Użyj anotacji `@Override`, a kompilator poinformuje Cię o błędzie

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class Example2Solution extends WindowAdapter {
    // Błąd komplikacji!
    @Override
    public void windowClose(WindowEvent e) {
        // Zakończ
        System.exit(0);
    }
}
```

Porównywanie typów prostych – operator ==

- Operatorem porównania dla typów prostych i referencji w Javie jest `==` a nie `=`
- Jednym ze sposobów ochrony przed tym błędem jest umieszczanie stałej po lewej stronie operatora, np.:

```
if (false == isReady) ...
if (MAX_AMOUNT >= currentAmount) ...
```

Porównywanie obiektów – metoda *equals*

- Porównywanie obiektów w Javie odbywa się tylko i wyłącznie za pomocą metody *equals*

```
if ("abc".equals(myString)) ...
if (person1.equals(person2)) ...
```

Nierozumienie sposobu przekazywania zmiennych do metod

- W Javie wszystkie parametry są przekazywane przez **wartość**, czyli parametry są kopiowane
- **Ale!** kopią referencji jest referencja do tego samego obiektu, a więc mamy dostęp do oryginalnego obiektu
- Należy uważać na klasy typu *immutable* takie jak *String*, *Integer*, *Double* i pozostałe typy opakowujące
- Mają charakter stały – nie umożliwiają zmiany swojej zawartości

Dławienie wyjątków

- Brak obsługi wyjątku!
 - powoduje ukrywanie błędów – niezwykle trudne do wykrycia
 - aplikacja pozornie działa dalej, ale działa nieprawidłowo!

```
import java.io.*;
public class Example3 {
    public static void main(String[] args) {
        File file = new File("file.txt");
        FileWriter fileWriter = null;
        try {
            fileWriter = new FileWriter(file);
            // ...
        } catch (IOException e) {
            // nic nie rob
        } finally {
            try {
                fileWriter.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Dławienie wyjątków – Rozwiążanie

- Jeśli nie ma innego pomysłu – należy przynajmniej użyć metody *printStackTrace*

```
import java.io.*;
public class Example3 {
    public static void main(String[] args) {
        File file = new File("file.txt");
        FileWriter fileWriter = null;
        try {
            fileWriter = new FileWriter(file);
            // ...
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                fileWriter.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Pomyłki w indeksowaniu tablic

- Tablice w Javie są indeksowane od zera!

```
public class Example4 {  
    public static void main(String[] args) {  
        // tablica zlozona z 3 ciagow znakowych  
        String[] strArray = new String[3];  
  
        // Pierwszy element jest pod indeksem 0  
        strArray[0] = "Pierwszy";  
  
        // Drugi element jest pod indeksem 1  
        strArray[1] = "Drugi";  
  
        // Trzeci element jest pod indeksem 2  
        strArray[2] = "Trzeci";  
    }  
}
```

Brak konstruktora domyślnego

- Stworzenie własnego konstruktora powoduje, że konstruktor domyślny nie jest generowany
- Wiele bibliotek wymaga istnienia konstruktora bezargumentowego
- Kiedy tworzysz konstruktor, upewnij się, czy nie powinieneś dodać do klasy konstruktora bezargumentowego

Pomyłki związane z konwencją kodowania

- Szczególnie ważne są konwencje związane ze specyfikacją *JavaBeans*
- Rozpoczynaj nazwy metod małymi literami
- Rozpoczynaj nazwy klas dużymi literami
- Metody dostępowe rozpoczynaj prefiksami: *get*, *set*, *is*
- Uważaj na notację wielbłędzią
- Literówki w nazwach mogą spowodować nieprawidłowe działanie bibliotek opierających się o te konwencje
- **Na szczęście wiele środowisk programistycznych informuje o tego typu błędach**

NullPointerException

- Najczęściej pojawiający się błąd
- Wynika z niedokładnie przemyślanego kodu
- Sprawdzaj parametry metod
- Upewnij się, że inicjujesz odpowiednio zmienne referencyjne

Podsumowanie

- Najczęściej popełniane błędy przez programistów Javy

Plan szkolenia

9

Zaawansowane kolekcje i we/wy

10

Biblioteka standardowa Java SE

11

Internacjonalizacja w języku Java

12

Programowanie wielowątkowe

13

Tworzenie interfejsu graficznego

14

Programowanie baz danych z użyciem JDBC API

Plan modułu

9

Zaawansowane kolekcje i we/wy

- sortowanie kolekcji i inne algorytmy
- filtrowanie strumieni
- klasa Scanner
- formatowanie wyjścia

Sortowanie kolekcji – interfejs Comparable

- Klasy implementujące interfejs *Comparable* pozwalają na porównywanie ze sobą dwóch obiektów i szeregowanie ich w określonej kolejności

```
public int compareTo(Object obj);
```

-
- Listy i tablice implementujące ten interfejs można sortować przy pomocy metody *Collections.sort()*

Sortowanie kolekcji – interfejs Comparable<T>

- Interfejs *Comparable* występuje w Javie SE 5.0+ również w wersji generycznej *Comparable<T>*
 - wtedy należy zaimplementować metodę:

```
public int compareTo(T obj);
```

Przykład – bez użycia typów generycznych

```
public class Person implements Comparable {  
    private String name;  
    private String secondName;  
  
    // sortowanie najpierw po nazwisku, a potem po imieniu  
    @Override  
    public int compareTo(Object o) {  
        Person person = (Person)o;  
        int secondNameComparison  
            = secondName.compareTo(person.secondName);  
        if (secondNameComparison != 0) {  
            return secondNameComparison;  
        }  
        return name.compareTo(person.name);  
    }  
    // ...  
}
```

Przykład – z użyciem typów generycznych

```
public class Person implements Comparable<Person> {
    private String name;
    private String secondName;

    // sortowanie najpierw po nazwisku, a potem po imieniu
    @Override
    public int compareTo(Person person) {
        int secondNameComparison
            = secondName.compareTo(person.secondName);
        if (secondNameComparison != 0) {
            return secondNameComparison;
        }
        return name.compareTo(person.name);
    }
    // ...
}
```

Sortowanie z użyciem compareTo

```
Person person1 = new Person("Jan", "Kowalski");
Person person2 = new Person("Piotr", "Kaminski");
Person person3 = new Person("Aleksander", "Wielki");

List<Person> persons = new ArrayList<Person>();
persons.add(person1);
persons.add(person2);
persons.add(person3);

// sortowanie wg metody compareTo
Collections.sort(persons);
```

Sortowanie kolekcji – interfejs Comparator

- Interfejs *Comparable* możemy implementować tylko raz
- Jeśli chcemy jedną kolekcję sortować na kilka różnych sposobów, należy tworzyć nowe obiekty implementujące interfejs *Comparator* i przekazywać je do metody *sort()*

```
int compare(Object o1, Object o2);  
boolean equals(Object obj);
```

Sortowanie kolekcji – interfejs Comparator<T>

- Interfejs *Comparator* w Javie SE 5.0+ istnieje również w wersji generycznej *Comparator<T>*

```
int compare(T o1, T o2);  
boolean equals(Object obj);
```

Implementacja komparatorów

```
import java.util.Comparator;

// porównywanie po imieniu
class PersonByNameComparator implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getName().compareTo(o2.getName());
    }
}
```

```
import java.util.Comparator;

// porównywanie po nazwisku
class PersonBySecondNameComparator implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getSecondName().compareTo(o2.getSecondName());
    }
}
```

Sortowanie z użyciem komparatorów

```
Person person1 = new Person("Jan", "Kowalski");
Person person2 = new Person("Piotr", "Kaminski");
Person person3 = new Person("Aleksander", "Wielki");

List<Person> persons = new ArrayList<Person>();
persons.add(person1);
persons.add(person2);
persons.add(person3);

// sortowanie po imieniu
Collections.sort(persons, new PersonByNameComparator());

// sortowanie po nazwisku
Collections.sort(persons, new PersonBySecondNameComparator());
```

Inne przykłady metod klasy użytkowej java.util.Collections

- Wyszukiwanie pozycji danego elementu na liście, kolekcja musi być posortowana

```
static int binarySearch(List list, Object key)
```

- Zastąpienie wszystkich elementów listy danym obiektem

```
static void fill(List list, Object obj)
```

- Odwrócenie kolejności elementów w liście

```
static void reverse(List list)
```

- Losowa kolejność elementów na liście

```
static void shuffle(List list)
```

Strumienie – przypomnienie

- Strumienie binarne
 - *InputStream*
 - *OutputStream*
- Służą przesyłaniu danych surowych, które mogą być następnie poddawane interpretacji
- Strumienie tekstowe
 - *Reader*
 - *Writer*
- Służą przesyłaniu danych tekstowych (opartych o *char*)
- Zapewniają odpowiednią konwersję znaków między stronami kodowymi węzła wejściowego i węzła wyjściowego

Filtrowanie danych

- Strumienie stanowią wygodną formę przekazywania danych, ale również wygodną formę do prowadzenia przekształceń danych
- Na strumień można nałożyć filtr (dekorator), który jest klasą pochodzącą od strumienia, ale uzupełnia go o funkcję przekształcania danych
- Filtry można wzajemnie na siebie nakładać, co daje możliwości złożonego przekształcania danych
- Najczęstsze zastosowania filtrów to:
 - buforowanie danych
 - przekształcanie danych do odpowiednich typów
 - kompresja lub szyfrowanie danych

Typowe filtry

- *DataInputStream*
- przekształcenie surowych danych binarnych w typy podstawowe
 - *readBoolean*
 - *readInt*
 - *readLong*
 - *readDouble*
 - *readUTF*
- *DataOutputStream*
- przekształcenie typów podstawowych w surowe dane binarne
 - *writeBoolean*
 - *writeInt*
 - *writeLong*
 - *writeDouble*
 - *writeUTF*

Korzystanie z plików

- Współpraca z plikami wymaga otwarcia strumienia związanego z plikiem
- Przykład:

```
FileWriter fw = new FileWriter("output.txt");
fw.write("Ala ma kota");
fw.close();
```

Korzystanie z plików – filtry buforujące

- We współpracy z plikami w celu poprawy efektywności stosuje się filtry buforujące:

```
BufferedWriter bw = new BufferedWriter(  
    new FileWriter("output.txt"));
```

- Dostarczają one między innymi metodę *newLine*:

```
bw.newLine();
```

Filtryle umożliwiające pracę z plikami zip

- Java SE zawiera wsparcie dla pracy z plikami zip
- Odpowiednikiem klasy *File* jest klasa *ZipFile*

```
public static void readZip() throws IOException {
    ZipFile file = new ZipFile("mod01.zip");

    for (Enumeration entries = file.entries();
         entries.hasMoreElements();) {
        String zipEntryName
            = ((ZipEntry) entries.nextElement()).getName();
        System.out.println(zipEntryName);
    }
}
```

Filtryle umożliwiające pracę z plikami zip

- Aby stworzyć archiwum zip musimy użyć filtra *ZipOutputStream*, którym opakowujemy strumień źródłowy
- Każdy nowy element dodajemy do archiwum poprzez użycie metody *putNextEntry*
- Od momentu wywołania tej metody bajty wysyłane do strumienia zip są przypisane do tego elementu
- Do *ZipOutputStream* zapisujemy dane jak do innych strumieni
- Należy pamiętać o zamknięciu elementu archiwum (*closeEntry*) i zamknięciu strumienia (*close*)

Kompresja kilku plików do archiwum zip

```
public static void createZip() throws IOException {
    // Nazwy plikow do zarchiwizowania
    String[] filenames = new String[] {"filename1", "filename2"};

    // Bufor do odczytu plikow
    byte[] buf = new byte[1024];

    // Stworzenie strumienia ZIP
    String outfile = "outfile.zip";
    ZipOutputStream out = new ZipOutputStream(
        new FileOutputStream(outfile));
}
```

- c.d. na następnym slajdzie

Kompresja kilku plików do archiwum zip

```
// Kompresja plikow
for (int i = 0; i < filenames.length; i++) {
    FileInputStream in = new FileInputStream(filenames[i]);

    // Dodaj nowy element do archiwum ZIP
    out.putNextEntry(new ZipEntry(filenames[i]));

    // Skopiuj bajty z plikow do archiwum
    int len;
    while ((len = in.read(buf)) > 0) {
        out.write(buf, 0, len);
    }

    // Zamknij element archiwum ZIP
    out.closeEntry();
    in.close();
}

// Zamknij strumien ZIP
out.close();
}
```

Serializacja danych

- Mechanizm serializacji danych umożliwia uszeregowanie danych obiektu do strumienia
 - Zapisanie obiektu do strumienia:

```
ObjectOutputStream oos = new ObjectOutputStream(  
                    new FileOutputStream("time.ser"));  
oos.writeObject(new Date());  
oos.close();
```

- Odczytanie stanu obiektu:

```
ObjectInputStream ois = new ObjectInputStream(  
                    new FileInputStream("time.ser"));  
Date date = (Date) ois.readObject();  
ois.close();
```

Serializacja danych

- Podczas serializacji danych zapamiętywana jest cała struktura danych obiektu
 - również wszystkie obiekty, do których referencje są trzymane jako składniki klasy (np. całe kolekcje)
 - aby operacja mogła być przeprowadzona serializowane obiekty (również te zagnieżdżone) muszą implementować znacznikowy interfejs *Serializable*
- Dane które nie mogą (strumienie, wątki) lub nie mają być serializowane muszą być oznaczone jako *transient*

```
public class Container implements Serializable {  
    private String name;  
    private transient String password;  
    // ...  
}
```

Serializacja danych

- Pierwsza klasa nadziedziona, która nie implementuje interfejsu *Serializable*, powinna mieć konstruktor bezargumentowy, aby serializacja się powiodła
- W klasie implementującej interfejs *Serializable* można zadeklarować:

```
private static final long serialVersionUID = 1234L;
```

-
- umożliwia wersjonowanie klas, dzięki czemu mechanizm deserializacji może stwierdzić, czy deserializowany obiekt jest zgodny z dostępna wersją klasy
 - jeśli brak jest zgodności tej wartości, zostanie wyrzucony wyjątek *InvalidClassException*

Klasa Scanner

- Pozwala na wczytywanie danych z łańcucha tekstowego, konsoli, bądź dowolnego innego strumienia danych
- Implementuje prosty skaner tekstu, wykorzystujący wyrażenia regularne do wyszukiwania typów prostych w źródle danych
- *Scanner* dzieli dane źródłowe na pojedyncze tokeny poprzez odnajdywanie znaków separatora
- Separator można ustawić używając metody *useDelimiter*
- Do pobierania danych posiada metody: *next*, *nextInt*, *nextFloat*, *nextByte*, *nextBoolean*, itd.

Klasa Scanner

- Przykład użycia:

```
Scanner sc = new Scanner(System.in);
String param = sc.next();
int i = sc.nextInt();
sc.close();
```

Formatowanie wyjścia

- Java udostępnia programistie nowe możliwości formatowania wyjścia, a to za sprawą implementacji w klasie *PrintStream* metod *printf* i *format*

```
printf(String format, Object ... args);  
format(String format, Object ... args);
```

gdzie:

format – łańcuch reprezentujący sposób rozmieszczenia poszczególnych podciągów w formatowanym tekście

args – dane, które mają być umieszczone w łańcuchu *format*

Formatowanie wyjścia

- Do oznaczania miejsc, w których mają być wstawione argumenty oraz ich typów i ewentualnie opcji wstawiania służą następujące oznaczenia:
 - %b – wartość typu *boolean* lub *null*
 - %c – znak w standardzie *Unicode*
 - %d – liczba dziesiętna
 - %f – liczba zmiennoprzecinkowa
 - %s – ciąg znaków
- Przykład użycia:

```
System.out.printf("Uzytkownik %s ma ID %d", "Nowak", 1);
```

Klasa NumberFormat

- Klasa dostarcza interfejs do formatowania danych liczbowych w zależności od ustawionej lokalizacji
- Kod staje się niezależny od lokalnej reprezentacji liczb zmiennoprzecinkowych
 - `getNumberInstance` – pobiera format zwykłych liczb
 - `getIntegerInstance` – pobiera format typu *Integer*
 - `getCurrencyInstance` – pobiera format walut
 - `getPercentInstance` – pobiera format procentów

Klasa NumberFormat

- Przykład zastosowania:

```
NumberFormat nf = NumberFormat.getPercentInstance();  
double procent = 0.23;  
System.out.println(nf.format(procent));
```

Klasa DateFormat

- Klasa dostarcza interfejs do formatowania daty lub czasu w zależności od ustawionej lokalizacji
- Kod staje się niezależny od lokalnej reprezentacji daty i czasu
- Można wyświetlać nazwy miesięcy i dni tygodnia w języku zdefiniowanym w danej lokalizacji
 - `getDateInstance` – pobiera format daty
 - `getTimeInstance` – pobiera format czasu
 - `getDateTimeInstance` – pobiera format daty i czasu
- Wszystkie powyższe metody fabryki mogą przyjmować jako argument styl wyświetlania oraz lokalizację, której format mają pobrać
- Dostępne style:
 - `DateFormat.SHORT` — 29.12.05
 - `DateFormat.MEDIUM` — 2005-12-29
 - `DateFormat.LONG` — 29 grudzień 2005
 - `DateFormat.FULL` — czwartek, 29 grudzień 2005

Klasa SimpleDateFormat

- Klasa *SimpleDateFormat* pozwala użytkownikowi na formatowanie i przetwarzanie dat w lokalnym systemie zapisu, lub w systemie zdefiniowanym przez samego użytkownika
- Pozwala na:
 - formatowanie: data → tekst
 - przetwarzanie: tekst → data
- Użytkownik może wybrać zdefiniowany wcześniej format wyświetlania lub też dostosować go do swoich potrzeb, bądź nawet stworzyć własny sposób prezentacji daty
- Klasa ta wymaga wzorca wyświetlania. Wzorzec taki składa się z liter o odpowiednim znaczeniu i ciągów znaków ograniczanych przez ' ', które są traktowane dosłownie

Klasa SimpleDateFormat

- Przykład zastosowania:

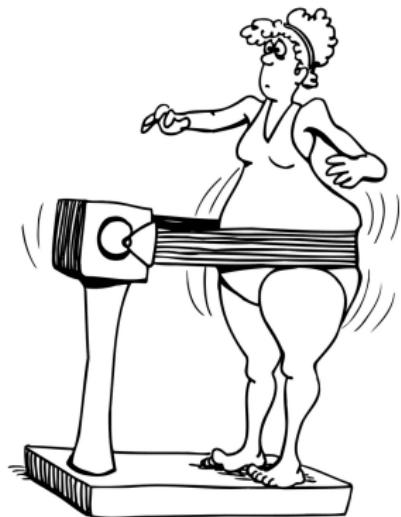
```
SimpleDateFormat formatter;  
formatter = new SimpleDateFormat("Dzis jest' EEE d MMM yy");  
Date today = new Date();  
System.out.println(formatter.format(today));
```

Pytania sprawdzające

- Podaj sposoby sortowania kolekcji/tablic
- Jakie możliwości oferuje klasa *Collections*?
- Podaj przykłady klas dekoratorów
- Jakie klasy mogą być przydatne przy pracy z archiwami zip?
- Na czym polega serializacja?
- Jakie wymagania muszą być spełnione, aby serializacja się powiodła?
- Jakie zastosowanie ma klasa *Scanner*?
- Podaj przykłady klas formatujących dane

Ćwiczenia

- Ćwiczenie 9.1:
*program TravelOffice:
wykorzystanie mechanizmu serializacji
współpraca z plikami
wykorzystanie systemowych klas zapisu
daty
formatowanie wejścia/wyjścia
z wykorzystaniem klas systemowych*



Plan szkolenia

9

Zaawansowane kolekcje i we/wy

10

Biblioteka standardowa Java SE

11

Internacjonalizacja w języku Java

12

Programowanie wielowątkowe

13

Tworzenie interfejsu graficznego

14

Programowanie baz danych z użyciem JDBC API

Plan modułu

10

Biblioteka standardowa Java SE

- logowanie (Logging API)
- wyrażenia regularne
- kilka krótkich przykładów

Logowanie – Logging API

- **Logowanie lub dziennik zdarzeń** to technika zapisu informacji na temat przebiegu wykonania aplikacji lub systemu
- Służy rejestracji zdarzeń i operacji, które nastąpiły wskutek tego przebiegu
- Polega na zapisie informacji do pliku, bazy danych, na konsolę lub w innym miejscu docelowym, które wskaże programista

Logowanie – Logging API

- Standardowy mechanizm obsługujący logowanie to *Logging API* w Javie SE
- Istnieją alternatywy (bardziej rozbudowane)
 - np. *Apache Logging Services (log4j)*
<http://logging.apache.org/>

Logowanie – Logging API

- Logowanie może mieć wiele zastosowań
 - forma dostarczania informacji administratorowi o tym, co się dzieje w systemie
 - alternatywa lub uzupełnienie systemów debugujących
 - szczególnie przydatne w systemach wielowątkowych, gdy trudno debugować jednocześnie kilka wątków
 - narzędzie zastępujące `System.out.println`, dające większą kontrolę nad sposobem zapisu informacji

Logowanie – Logging API

- Komunikaty należy zapisywać używając obiektu klasy *Logger*

```
private static Logger logger
        = Logger.getLogger("pl.altkom.jpr");
// ...
logger.info("dzieje sie cos istotnego!");
```

- Pojedynczy obiekt *Logger* ma tekstową nazwę, dzięki której można go odnaleźć
- Najczęściej jako nazwy używa się pakietu lub pełnej nazwy klasy

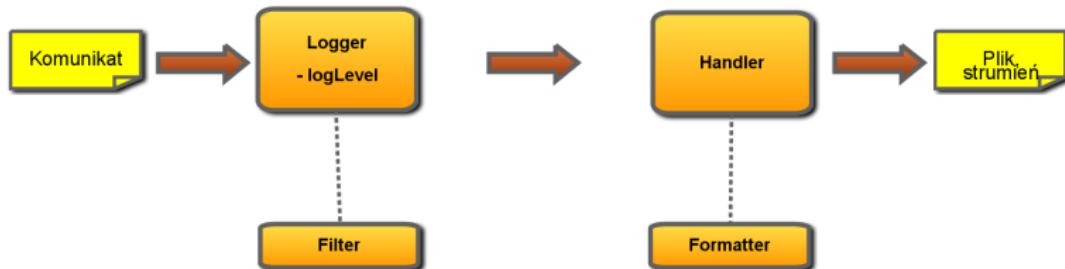
Logowanie – Logging API

- Obiekt klasy *Handler* zapisuje komunikaty w wybrane przez programistę miejsce (np. konsola, plik)

```
private static FileHandler fh
    = new FileHandler("mylog.txt");
// ...
logger.addHandler(fh);
```

- Przykłady dostępnych implementacji:
 - StreamHandler* – zapis komunikatów do dowolnego strumienia
 - ConsoleHandler* – zapis komunikatów na *System.err*
 - FileHandler* – zapis do plików (także rotacyjnie)
 - SocketHandler* – zapis do portu TCP

Logging API – ogólny schemat



Logowanie – Logging API

- Każdemu obiektowi typu *Logger* możemy nadać poziom zapisu informacji
- Wszystkie komunikaty o priorytecie niższym od poziomu *Loggera* będą ignorowane
- Dostępne poziomy logowania – od najwyższego priorytetu do najniższego: *SEVERE*, *WARNING*, *INFO*, *CONFIG*, *FINE*, *FINER*, *FINEST*
- Aby wyświetlić komunikat na danym poziomie używamy odpowiednich metod obiektu *Logger*

```
logger.info("komunikat");
logger.fine("komunikat");
logger.warning("komunikat");
```

Logging API – Poziomy logowania

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggerLevels {
    public static void main(String[] args) {
        Logger logger = Logger.getLogger("pl.altkom.jpr");
        logger.setLevel(Level.INFO);

        // ze wzgledu na poziom loggera zostana zignorowane
        logger.finest("Komunikat na poziomie FINEST");
        logger.config("Komunikat na poziomie CONFIG");

        // ze wzgledu na poziom loggera zostana wyswietlone
        logger.info("Komunikat na poziomie INFO");
        logger.severe("Komunikat na poziomie SEVERE");
    }
}
```

Logging API – Przykład użycia

```
import java.io.IOException;
import java.util.logging.*;

public class LoggerSample {
    private static Logger logger
        = Logger.getLogger("pl.altkom.jpr");

    public static void main(String[] args) {
        logger.setLevel(Level.CONFIG);
        LoggerSample loggerSample = new LoggerSample();
        loggerSample.start();
    }

    public void start() {
        logger.info("Uruchamianie aplikacji ...");
        try {
            // odczyt plikow konfiguracyjnych
            logger.config("Odczytano konfiguracje z pliku ...");
        } catch (IOException e) {
            logger.severe("Blad odczytu pliku konfiguracyjnego ...");
            return;
        }
        logger.finest("Odczyt pliku konfiguracyjnego powiodl sie");
        // ...
    }
}
```

Wyrażenia regularne

- Począwszy od J2SE wersja 1.4, udostępniono pakiet `java.util.regex` pozwalając tym samym na wykorzystanie w programach wyrażeń regularnych
- W celu skorzystania z wyrażenia regularnego, należy utworzyć łańcuch reprezentujący to wyrażenie, a następnie wkomplilować go do wzorca
- Tak przygotowany wzorzec można wykorzystać do stworzenia obiektu typu *Matcher*, który dokona wyszukania wyrażenia w podanym ciągu

Wyrażenia regularne

- Zbiór znaków oznaczamy – []
- Zdefiniowane klasy znaków:
 - . dowolny znak
 - \d cyfra: [0-9]
 - \D inny niż cyfra: [^0-9]
 - \s znak odstępu: [\t\n\x0B\f\r]
 - \S inny niż znak odstępu: [^\s]
 - \w znak: [a-zA-Z_0-9]
 - \W inny niż znak: [^\w]
 - () definicja sekwencji znaków

Wyrażenia regularne

- Zdefiniowane kwantyfikatory:

- | | |
|-----------------------------|---|
| ● XX? | raz lub wcale |
| ● XX* | zero lub więcej |
| ● XX ⁺ | raz lub więcej |
| ● XX{ <i>n</i> } | dokładnie <i>n</i> razy |
| ● XX{ <i>n</i> , <i>m</i> } | przynajmniej <i>n</i> razy |
| ● XX{ <i>n</i> , <i>m</i> } | przynajmniej <i>n</i> razy, lecz nie więcej niż <i>m</i> razy |

Wyrażenia regularne

- Obiekt wzorca można wykorzystać wielokrotnie do poszukiwania w wielu różnych ciągach

```
Pattern p = Pattern.compile("k*t");
Matcher m = p.matcher("kooooooooooooootek");
boolean b = m.matches();
```

- W przypadku pojedynczego poszukiwania można użyć konstrukcji:

```
boolean b = Pattern.matches("k*t", "kooooooooooooootek");
```

Wyrażenia regularne (podziałłańcuchów)

- Wyrażenia regularne można zastosować przy podziale tekstu wokół podanego wzorca

```
Pattern p = Pattern.compile("[,\\s]+");
String[] result = p.split("one,two, three four , five");
for (String i : result)
    System.out.println("|" + i + "|");
```

- Taki sam efekt można uzyskać przy wykorzystaniu metody *String.split(String regex)*, podając jej jako argument wyrażenie regularne

```
String str = "Raz1dwa124trzy";
String[] d1 = str.split("d"); // podzial wokol litery d
String[] d2 = str.split("\\d"); // podzial wokol cyfry
```

Serializacja do XML – pakiet `java.beans`

```
import java.beans.XMLEncoder;
import java.io.*;

public class SerializationToXML {
    public static void main(String[] args) {
        // ...
        XMLEncoder encoder = null;
        try {
            // Serializacja XML
            encoder = new XMLEncoder(
                new BufferedOutputStream(
                    new FileOutputStream("outfilename.xml")));
            // serializujemy obiekt o referencji o
            encoder.writeObject(o);
        } catch (FileNotFoundException e) {
            // ...
        } finally {
            if (encoder != null) encoder.close();
        }
    }
}
```

Operowanie na liczbach (obliczenia księgowe i naukowe) – pakiet `java.math`

```
import java.math.BigDecimal;

public class BigDecimalSample {
    public static void main(String[] args) {
        // Stworzenie liczby na bazie ciągu znakowego
        BigDecimal bd1 = new BigDecimal("123456789.0123456890");

        // Stworzenie liczby na bazie typu long
        BigDecimal bd2 = BigDecimal.valueOf(123L);

        bd1 = bd1.add(bd2);
        bd1 = bd1.multiply(bd2);
        bd1 = bd1.subtract(bd2);
        bd1 = bd1.divide(bd2, BigDecimal.ROUND_UP);
        bd1 = bd1.negate();
    }
}
```

Podział ciągu tekstowego na tokeny – java.util.StringTokenizer

```
import java.util.StringTokenizer;

public class TokenizerSample {
    public static void main(String[] args) {
        String aString = "word1 word2 word3";
        StringTokenizer parser = new StringTokenizer(aString);

        while (parser.hasMoreTokens()) {
            String token = parser.nextToken();
            // ....
        }
    }
}
```

Zdarzenia inicjowane w określonym czasie – java.util.Timer

- Zadanie jest uruchamiane w osobnym wątku, więc uwaga na konsekwencje współbieżności

```
import java.util.*;

public class TimerSample {
    public static void main(String[] args) {
        final int delayInMs = 10000;
        Date timeToRun = new Date(
            System.currentTimeMillis() + delayInMs);
        Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            public void run() {
                // Zadanie ...
            }
        }, timeToRun);
    }
}
```

Odczyt zawartości strony WWW – java.net.URL

```
import java.io.*;
import java.net.*;

public class WebReader {
    public static void main(String[] args) {
        try {
            // url z adresem strony
            URL url = new URL("http://hostname:80/index.html");

            // strumień do odczytu danych
            BufferedReader in = new BufferedReader(
                new InputStreamReader(url.openStream()));

            String str;
            while ((str = in.readLine()) != null) {
                // operacje na pojedynczych wierszach zwroconych danych
            }
            in.close();
        } catch (MalformedURLException e) {
            // ...
        } catch (IOException e) {
            // ...
        }
    }
}
```

Podsumowanie

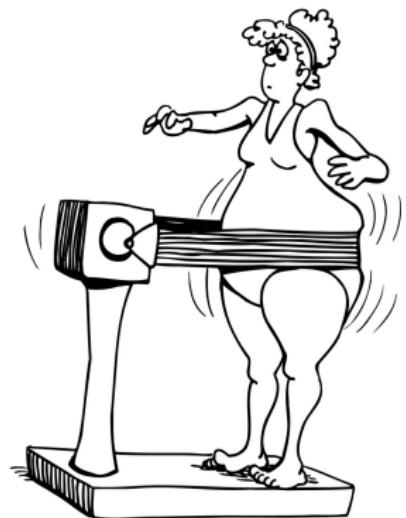
- Logowanie (Logging API)
- Wyrażenia regularne
- Kilka krótkich przykładów

Pytania sprawdzające

- Jakie są zastosowania logowania?
- Jaki jest ogólny schemat logowania?
- Do czego można zastosować wyrażenia regularne?
- Wymień kwantyfikatory stosowane w wyrażeniach regularnych
- Podaj przykłady znaków specjalnych używanych w wyrażeniach regularnych

Ćwiczenia

- Ćwiczenie 10.1:
użycie w aplikacji mechanizmów Java SE



Plan szkolenia

9

Zaawansowane kolekcje i we/wy

10

Biblioteka standardowa Java SE

11

Internacjonalizacja w języku Java

12

Programowanie wielowątkowe

13

Tworzenie interfejsu graficznego

14

Programowanie baz danych z użyciem JDBC API

Plan modułu

11

Internacjonalizacja w języku Java

- potrzeba wielojęzyczności
- obiekt typu Locale
- klasa ResourceBundle
- formatowanie

Potrzeba wielojęzyczności

- W dobie powszechniej globalizacji jednym z podstawowych wymagań wobec aplikacji jest obsługa wielojęzyczności
- W Javie istnieje szereg narzędzi wspierających prace z aplikacjami wielojęzycznymi
- Podstawową klasą jest *Locale*, przechowująca dane o parametrach regionalnych takich jak: kraj, język, wariant
- Istnieje wiele klas pomocniczych ułatwiających formatowanie danych w sposób odpowiadający zasadom panującym w danym regionie

Klasa Locale

- Obiekt klasy *Locale* zazwyczaj określa dwie informacje: kod języka i kod kraju
- Jest jeszcze dodatkowa informacja – kod wariantu, rzadziej używana np. w celu zróżnicowania zachowania na różnych platformach – UNIX, WINDOWS
- Tworzenie obiektu:

```
Locale plLocale = new Locale("pl", "PL");
```

- lub za pomocą stałych:

```
Locale japanLocale = Locale.JAPAN;
```

Przykłady wykorzystania klasy Locale

- Aby odczytać lub zmienić *Locale* na poziomie globalnym – dla maszyny wirtualnej, należy wywołać metodę *Locale.getDefault* lub *Locale.setDefault*
- Przykłady elementów Java SE, które są wrażliwe na ustawienia *Locale*
 - *DateFormat*
 - *NumberFormat*
 - *Collator* – sortowanie wrażliwe na ustawienia *Locale*
 - Nazwy stref czasowych
 - Symbole walut

Klasa ResourceBundle

- Służy do obsługi wielojęzycznych napisów
- Jest klasą abstrakcyjną z wieloma metodami pomocniczymi
- Dane przechowywane są na zasadzie **klucz = wartość**
- Pod tym samym kluczem dla różnych języków będzie inna wartość
- Jedną z możliwości jest napisanie własnej implementacji rozszerzającej *ListResourceBundle*, która obsłuży w odpowiedni sposób wymagane języki
- W praktyce najczęściej używa się plików do przechowywania informacji

Klasa ResourceBundle – pliki properties

- Teksty, które mają mieć różną formę dla różnych języków, przechowywane są w plikach tekstowych o rozszerzeniu *properties*, np.: *labels.properties*
 - w tym przypadku *labels* to nazwa bazowa
- Różne wersje językowe należy tworzyć dodając odpowiednie sufiksy do nazwy bazowej, np.:
 - *labels_fr_FR.properties*
 - *labels_en_GB.properties*
 - *labels_en_US.properties*

Klasa ResourceBundle – pliki properties

- Pliki mają następującą strukturę – każdy wiersz to para:
klucz = wartość
- Na przykład:
 - labels_en_GB.properties:

```
hello = Hello
goodbye = Goodbye
welcome.message = Welcome to the brave new world
```

- labels_pl_PL.properties:

```
hello = Czesc
goodbye = Do widzenia
welcome.message = Witaj w nowym, lepszym swiecie
```

Klasa ResourceBundle – pliki properties

- Aby pobrać wielojęzyczny tekst, należy najpierw uzyskać dostęp do odpowiedniej wersji językowej
- Na przykład:

```
ResourceBundle bundle  
        = ResourceBundle.getBundle("labels", locale);
```

- Następnie można pobrać tekst w danej wersji językowej:

```
bundle.getString("welcome.message");
```

Klasa MessageFormat

- Java SE umożliwia tworzenie szablonów tekstów, wrażliwych na *Locale*
- Założymy, że potrzebny jest szablon tekstowy o następującej strukturze:
*Dnia <tutaj_konkretna_data> z miasta
<nazwa_pierwszego_miasta> do miasta
<nazwa_drugiego_miasta> wyjedzie <ilość_autobusów>
autobusów*
- Szablon ten można zapisać w formie czytelnej dla klasy *MessageFormat*:

```
String template = "Dnia {1, date, long} z miasta {0} do"  
                  + " miasta {2} wyjedzie {3} autobusow";
```

Klasa MessageFormat

- W szablonie parametry szablonu umieszcza się w nawiasach klamrowych
- W przypadku dat, czasu i liczb można dodać dodatkowe parametry
- Szczegóły w Java API
- Szablon można zastosować do obiektu klasy *MessageFormat*

```
MessageFormat messageFormat = new MessageFormat(template);
```

- Używając metody *MessageFormat.format* i dostarczając parametry poprzez tablicę obiektów, uzyskujemy sformatowany tekst

Klasa MessageFormat – Przykład

```
String template = "Dnia {1, date, long} z miasta {0} do"
    + " miasta {2} wyjedzie {3} autobusow";

// szablon mozna rowniez pobrac z ResourceBundle
// String template = resourceBundle.getString("template");

MessageFormat messageFormat = new MessageFormat(template);
Object[] arguments = new Object [] {
    resourceBundle.getString("first.city"),
    new Date(),
    resourceBundle.getString("second.city"),
    8L
};
```

Klasa ChoiceFormat

- Klasa *ChoiceFormat* została stworzona w celu obsługi formatowania ciągów znakowych w zależności od pewnych warunków
- Świeście się nadaje do formatowania różnych odmian tego samego ciągu znakowego w liczbie mnogiej
- Aby skorzystać z tej klasy należy określić przedziały liczbowe, w których słowo ma różne formy

```
double[] limits = {0, 1, 2, 5};
```

- Następnie należy podać formy tekstu, które mają być wyświetlane w zadanym przedziale

```
String [] wyjedzieForms  
= {"wyjedzie", "wyjedzie", "wyjada", "wyjedzie"};
```

Klasa ChoiceFormat

- Mając dwa elementy: przedziały i teksty, należy stworzyć obiekt klasy *ChoiceFormat*

```
ChoiceFormat wyjedzieChoiceFormat  
        = new ChoiceFormat(limits, wyjedzieForms);
```

- Tak utworzony format należy dołączyć do obiektu klasy *MessageFormat*

```
Format[] formats = {null, null, null,  
                    wyjedzieChoiceFormat,  
                    null,  
                    autobusChoiceFormat};  
  
messageFormat.setFormats(formats);
```

Zastosowanie klasy ChoiceFormat

```
// mozna pobrac z ResourceBundle
String template = "Dnia {1, date, long} z miasta {0} do"
                  + " miasta {2} {3} {4} {5}";
double[] limits = {0, 1, 2, 5}; // granice przedzialow
String[] wyjedzieForms
    = {"wyjedzie", "wyjedzie", "wyjada", "wyjedzie"};
String[] autobusForms
    = {"autobusow", "autobus", "autobusy", "autobusow"};
ChoiceFormat wyjedzieChoiceFormat
    = new ChoiceFormat(limits, wyjedzieForms);
ChoiceFormat autobusChoiceFormat
    = new ChoiceFormat(limits, autobusForms);
```

Zastosowanie klasy ChoiceFormat

```
MessageFormat messageFormat = new MessageFormat(template);
Format[] formats = {null, null, null,
                     wyjedzieChoiceFormat,
                     null,
                     autobusChoiceFormat};
messageFormat.setFormats(formats);

for (int i = 0; i < 10; i++) {
    Object[] arguments = new Object [] {
        "Lodzi",
        new Date(),
        "Warszawy",
        i,
        i,
        i};

    System.out.println(messageFormat.format(arguments));
}
```

Podsumowanie

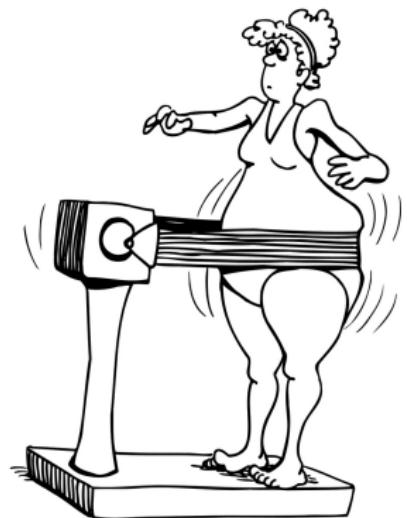
- Potrzeba wielojęzyczności
- Obiekt typu *Locale*
- Klasa *ResourceBundle*
- Formatowanie

Pytania sprawdzające

- Jakie dane można podać konstruktorom klasy *Locale*?
- Jaką strukturę zawartości mają pliki własności?
- Jak należy nazwać pliki własności dla różnych języków?
- Jaką rolę pełni klasa *MessageFormat*?
- Do czego jest przydatna klasa *ChoiceFormat*?

Ćwiczenia

- Ćwiczenie 11.1:
*wprowadzenie w aplikacji elementów
wielojęzyczności*



Plan szkolenia

9 Zaawansowane kolekcje i we/wy

10 Biblioteka standardowa Java SE

11 Internacjonalizacja w języku Java

12 Programowanie wielowątkowe

13 Tworzenie interfejsu graficznego

14 Programowanie baz danych z użyciem JDBC API

Plan modułu

12

Programowanie wielowątkowe

- tworzenie aplikacji wielowątkowych
- cykl życia wątku
- stany wątków
- ochrona danych
- współpraca wątków
- kolekcje i wielowątkowość

Czym jest wątek

- Przepływ sekwencji sterowania w programie, posiadający własny stos wywołań
- Wszystkie wątki wykonywane w ramach jednego procesu mają dostęp do tej samej przestrzeni adresowej
- Każda aplikacja rozpoczyna swoje działanie poprzez wątek główny (*main thread*)
 - wątek główny może powołać i rozpoczęć wykonywanie innego wątku
 - w oparciu o ten sam kod i dane może być powołanych wiele wątków
- W rzeczywistym środowisku wątki konkurują między sobą o dostęp do fizycznych zasobów maszyny
 - pozornie działają równolegle
 - dają iluzję "bycia" w wielu punktach programu jednocześnie

Potrzeba stosowania wątków

- Możliwość równoległego wykonywania akcji w ramach systemu
 - nadzorowanie działania innych obszarów systemu
 - wykonywanie długotrwałych zadań w tle
 - realizacja animacji, odświeżania interfejsu użytkownika, zbierania zdarzeń
- Przy stosowaniu wątków pojawia się utrudnienie związane z zapewnieniem integralności współdzielonych danych

Tworzenie wątków – sposób 1

- Aby powołać wątek do życia należy:
 - przygotować kod i dane na których będzie operował wątek
 - może to być dowolna klasa implementująca interfejs
java.lang.Runnable
 - klasa ta musi zaimplementować metodę *run* z ww. interfejsu
 - zestaw instrukcji wątku stanowi kod metody *run*
 - utworzyć obiekt typu *java.lang.Thread* wskazując jednocześnie na jakiej instancji *Runnable* będzie operować
- Mając przygotowany obiekt typu *Thread* z osadzoną odpowiednią instancją typu *Runnable*:
 - wywołanie metody *start* rozpoczyna cykl wykonywania wątku
 - wątek żyje tak długo, aż:
 - zakończy wykonywać kod metody *run* z interfejsu *Runnable*
 - zostanie przerwany

Powołanie wątku – sposób 1

```
public class TestThread implements Runnable {  
    // ...  
    public void run() {  
        while (!end) {  
            System.out.println("New Java on the block");  
            // ...  
        }  
    }  
    public static void main(String args[]) {  
        TestThread tt = new TestThread();  
        Thread t1 = new Thread(tt);  
        Thread t2 = new Thread(tt);  
        t1.start();  
        t2.start();  
        // ...  
    }  
}
```

Tworzenie wątków – sposób 2

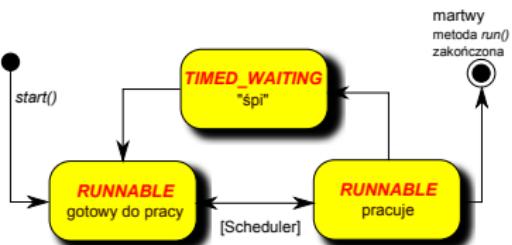
- Wątek można też utworzyć poprzez rozszerzenie klasy *Thread*
 - klasa *Thread* implementuje *Runnable*
 - zestaw instrukcji wątku stanowi kod nadpisanej metody *run* przez klasę dziedziczącą po *Thread*
- Dziedziczenie po *Thread* daje nieco prostszy kod
 - taka klasa nie może jednak rozszerzać innych klas biznesowych
 - czasami może zajść potrzeba nadpisania (ang. *override*) innych funkcji z klasy *Thread*
- Wykorzystanie interfejsu *Runnable* umożliwia lepsze:
 - zastosowanie w pełni podejścia obiektowego
 - osiągnięcie lepszej spójności klas
 - osiągnięcie większej elastyczności

Powołanie wątku – sposób 2

```
public class TestThread extends Thread {  
    // ...  
    public void run() {  
        while (!end) {  
            System.out.println("New Java on the block");  
            // ...  
        }  
    }  
    public static void main(String args[]) {  
        Thread t1 = new TestThread();  
        Thread t2 = new TestThread();  
        t1.start();  
        t2.start();  
        // ...  
    }  
}
```

Wykonywanie wątku

- Wątek po otrzymaniu sygnału `start` przechodzi do stanu *Runnable*
- Mechanizm Schedulera** zarządza wykonywaniem wątków – szereguje je, decydując kiedy, któremu przydzielić czas procesora
- Scheduler zarządza wątkami w sposób wywłaszczeniowy w oparciu o ich priorytety
 - wątek o wyższym priorytecie zazwyczaj wywłaszcza ten o niższym
 - Scheduler może wybrać wątek o niższym priorytecie, aby uniknąć zagłodzenia
- Mechanizm zarządzania wątkami jest oparty o rozwiązania z danej platformy operacyjnej
 - podział czasu
 - własna granulacja priorytetów



Wykonywanie wątków

- W **modelu wywłaszczeniowym** Scheduler decyduje o tym, kiedy wątek powinien uzyskać dostęp do zasobów – np. na podstawie priorytetów
- Wątki powinny zatem współpracować ze sobą zrzekając się czasu procesora
- Metody statyczne:

`yield` aktualny wątek zrzeka się zasobów

Scheduler wybiera z puli wątków ten, który zajmie jego miejsce

`sleep` uśpienie aktualnego wątku na określony czas

następnie przesunięcie do puli wątków oczekujących na zasoby

Wykonywanie wątków – przykład

```
class MyThread extends Thread {  
    // ...  
    public void run() {  
        // ...  
        new Task().count();  
    }  
}
```

```
class Task {  
    public void count() {  
        // ...  
        try {  
            Thread.sleep(100);  
        } catch (InterruptedException e) { //... }  
    }  
}
```

Metody sterujące wątkami

Metoda *join*

określony wątek może poczekać na zakończenie działania innego wątku poprzez wywołanie na jego referencji metody *join*.
join może być wywołane z ograniczeniem czasowym

```
Thread t = new Thread(testThread);  
t.start();  
t.join(100);  
// aktualny watek czeka max. 100 ms na zakoñczenie watku t
```

Metoda *setPriority*

ustala priorytet wątków

dostępne wartości statyczne z klasy *java.lang.Thread*:

- *MIN_PRIORITY* (1)
- *NORMAL_PRIORITY* (5)
- *MAX_PRIORITY* (10)

Metody sterujące wątkami

Metoda <i>setDaemon</i>	ustawia wątek na demoniczny wątek demoniczny nie wpływa na zakończenie życia aplikacji czas życia aplikacji jest określony przez czas życia wszystkich wątków niedemonicznych
Metoda <i>isAlive</i>	określa stan wątku sprawdza, czy wykonano metodę <i>start</i> i czy wątek jest jeszcze wykonywany

Kończenie pracy wątku

- Przerwanie wykonywanych przez wątek operacji powinno być dokonane przez wyjście z metody *run*, np. przez zastosowanie flagi sterującej:

```
private boolean end = false;  
// ...  
public void run() {  
    while (!end) {  
        // ...  
    }  
}  
public void stopThread() {  
    end = true;  
}
```

Metoda *stop* | użycie tej metody może doprowadzić do nieprawidłowego zwolnienia zasobów
| metoda zwalnia monitor do obiektu i stąd dane mogą być pozostawione w niespójnej postaci

Kończenie pracy wątku

Metoda *interrupt* ustawia wątkowi status przerwania, który powinien być okresowo sprawdzany za pomocą metody *isInterrupted*, jeśli w wątku nie występują metody *sleep*, *wait* lub *join*

```
public static void main(String[] args)
    throws InterruptedException {
    Thread t = new MyThread();
    t.start();
    Thread.sleep(100);
    t.interrupt();
}
```

```
// Brak metod sleep i wait w metodzie
public void run() {
    while (!Thread.currentThread().isInterrupted())
        System.out.println("Hello");
}
```

Kończenie pracy wątku

Metoda *interrupt*

jeśli została wywołana dla wątku, który akurat był zablokowany wywołaniem metod *sleep*, *wait* lub *join*, to wygeneruje ona wyjątek *InterruptedException*

wyjątek ten należy przechwytywać podczas wywoływania tych metod

metoda nie wpływa bezpośrednio na zakończenie wątka, a jedynie przekazuje sygnał przerwania, który wątek może wykorzystać, aby zakończyć swoje działanie

```
public void run() {  
    try {  
        while(true) {  
            System.out.println("Hello");  
            Thread.sleep(1000);  
        }  
    } catch (InterruptedException e) {  
        // kod przerywający wątek  
    } finally { /* ewentualne sprzątanie po sobie */ }  
}
```

Stany wątków

- W jednej chwili czasowej wątek może być tylko w jednym ze stanów:
 - *NEW*
 - *RUNNABLE*
 - *BLOCKED*
 - *WAITING*
 - *TIMED_WAITING*
 - *TERMINATED*

Stany wątków

NEW	stan dla wątku, który jeszcze nie wystartował
RUNNABLE	wątek się wykonuje w wirtualnej maszynie, ale może oczekiwania na inne zasoby (procesor)
BLOCKED	wątek oczekuje na monitor do obiektu wątek potrzebuje monitora, aby wejść do metody synchronizowanej (bloku synchronizowanego)
TERMINATED	wątek zakończył swoje działanie – zakończono wykonywać metodę <i>run</i> – zakończenie wątka przez czynniki zewnętrzne

Stany wątków

WAITING

wątek znajduje się w tym stanie po wywołaniu jednej z metod:

- *Object.wait()*
- *Thread.join()*

wątek będąc w tym stanie oczekuje, aż inny wątek zakończy określone operacje:

- *wait* – czeka, aż inny wątek wywoła *notify/notifyAll*
- *join* – czeka na zakończenie innego wątka

TIMED_WAITING

wątek oczekuje przez określony czas

może znaleźć się w tym stanie po wykonaniu jednej z metod:

- *Thread.sleep(long)*
- *Object.wait(long)*
- *Thread.join(long)*

Potrzeba ochrony spójności danych

- Wątki utworzone na podstawie tej samej instancji obiektu *Runnable* operują na tych samych danych
 - różne wątki mogą uzyskać dostęp do tych samych obiektów
- Ponieważ wątki mogą być wywłaszczone może się zdarzyć, że wywłaszczony wątek pozostawił dane w pośredniej, nie integralnej postaci
- Zarządzanie ochroną danych jest jednym z trudniejszych aspektów programowania wielowątkowego (współbieżnego)

Integralność danych

- Scheduler zarządza pulą wątków przekazując im dostęp do zasobów
- Wątek wykonujący operacje w wybranej metodzie operując na danych klasy może zostać wywłaszczyony przez inny
 - wówczas pozostawia dane w postaci niespójnej
 - po przywróceniu do działania wywłaszczyony wątek spodziewa się stanu jaki pozostawił, co może prowadzić do dalszej degradacji integralności danych
- W obiekcie, który posiada metody operujące na danych klasy należy zwrócić uwagę na wszystkie operacje mogące zagrozić integralności danych

Integralność danych – przykład

```
public class Storage {  
    int indx;  
    char tab[];  
    // ...  
  
    void push(char c) {  
        tab[indx] = c;  
        indx++;  
    }  
  
    char pop() {  
        indx--;  
        return tab[indx];  
    }  
}
```

Ochrona obiektów

- W celu ochrony integralności obiektów można wykorzystać mechanizm monitora obiektu
- **Monitor obiektu** jest znacznikiem, który w danej chwili może być przejęty tylko przez jeden wątek
- Wątek posiadający monitor, jeśli zostanie wywłaszczyony nie oddaje monitora innemu wątkowi (lub obiekowi)
 - Scheduler nie wywłaszcza wątku posiadającego monitor do obiektu, na rzecz wątku, który potrzebuje monitora do tego obiektu
- Wejście w posiadanie znacznika odbywa się poprzez realizację bloku lub metody synchronizowanej

Ochrona danych

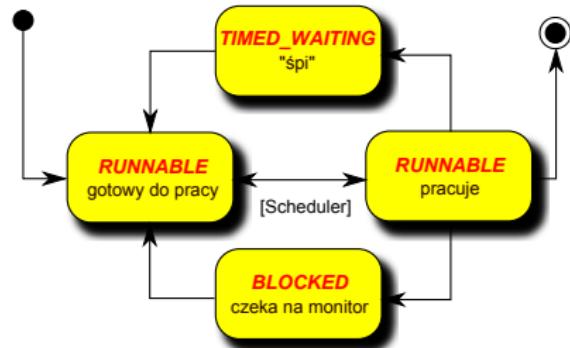
- Dane wrażliwe na zmianę muszą być chronione w bloku synchronizowanym
- W danej klasie blok synchronizowany może opierać się o dowolną referencję
- **Metoda synchronizowana** lub **blok synchronizowany** wymaga posiadania przez wątek monitora do obiektu na czas realizacji wszystkich operacji
- Należy ograniczać liczbę instrukcji w bloku synchronizowanym do najbardziej niezbędnych
 - blok synchronizowany nie jest wykonywany współbieżnie, dlatego stanowi "wąskie gardło" aplikacji
 - rozpoczęcie wykonywania bloku synchronizowanego lub metody synchronizowanej zabiera czas związany z przejęciem monitora

Ochrona danych – przykład

```
public class Storage {  
    int indx;  
    char tab[];  
    // ...  
  
    // metoda synchronizowana  
    synchronized void push(char c) {  
        tab[indx] = c;  
        indx++;  
    }  
  
    char pop() {  
        // blok synchronizowany  
        synchronized (this) {  
            indx--;  
            return tab[indx];  
        }  
    }  
}
```

Synchronizacja wątków

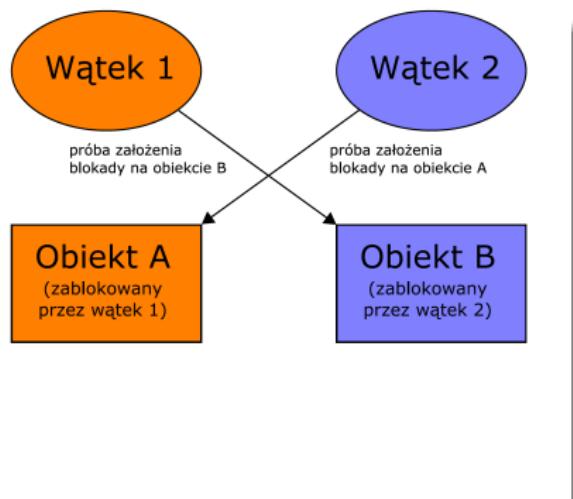
- Wątek potrzebujący monitora do obiektu, który został już pobrany przechodzi w stan oczekiwania na niego
- Gdy monitor stanie się dostępny Scheduler wybiera jeden z wątków z puli oczekujących na ten monitor, przekazuje mu monitor i ustawia go w kolejce *Runnable*
- Wszystkie dane wrażliwe powinny być deklarowane jako *private*
- Synchronizacja pogarsza wydajność aplikacji



Deadlock – zakleszczenie

- Sytuacja **zakleszczenia** występuje, gdy wątek posiadający monitor do obiektu oczekuje na dostęp do obiektu, którego monitor jest zajęty przez inny wątek. Ten inny wątek jednocześnie oczekuje na dostęp do obiektu zajętego przez pierwszy wątek

```
class A {  
    // ...  
    synchronized meth1() {  
        b.meth2();  
    }  
}  
  
class B {  
    // ...  
    synchronized meth2() {  
        a.meth1();  
    }  
}
```

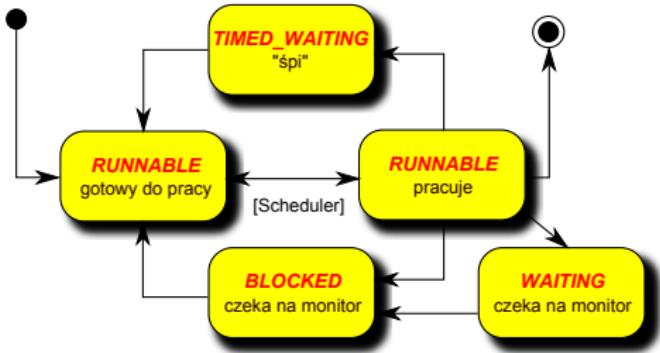


Deadlock – zakleszczenie

- Sytuacja zakleszczenia jest często trudna do wykrycia, gdyż zachodzi tylko w odpowiedniej zależności czasowej
- Poszukując potencjalnego zagrożenia zakleszczeniem należy rozpatrzyć wszystkie wywołania wymagające pobrania więcej niż jednego monitora

Współpraca wątków

- W sytuacji, gdy wątki kooperują w celu realizacji jednego zadania mogą wzajemnie powiadamiać się poprzez zestaw metod *wait/notify* z klasy *java.lang.Object*
 - typowy problem interakcji to **producent-konsument**
 - dostarczanie danych, wyników pośrednich, zleceń klienta
 - przetwarzanie danych, wyników, obsługa klienta
- Wykorzystanie powiadamiania poprzez metody *wait/notify* musi zostać zrealizowane w metodzie synchronizowanej lub bloku synchronizowanym
- Wątek oczekujący na powiadomienie jest "zawieszony" – zwraca monitor do obiektu



Współpraca wątków – przykład

```
synchronized void push(char c) {  
    while (indx == LIMIT) {  
        try {  
            wait();  
        } catch (InterruptedException e) { /* ... */ }  
    }  
    notify();  
    tab[indx] = c;  
    indx++;  
}  
  
synchronized char pop() {  
    while (indx == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) { /* ... */ }  
    }  
    notify();  
    indx--;  
    return tab[indx];  
}
```

Współpraca wątków

METODA	DZIAŁANIE
<i>wait</i>	przesuwa wątek do stanu oczekujących na powiadomienie wątek zwraca monitor do obiektu
<i>notify</i>	Scheduler wybiera wątek z puli wątków oczekujących na powiadomienie (<i>wait</i>) wątek ten jest przenoszony do puli wątków oczekujących na monitor
<i>notifyAll</i>	Scheduler przenosi wszystkie wątki oczekujące po <i>wait</i> do puli wątków oczekujących na monitor do obiektu

- Dobra praktyka mówi, aby wątek powiadomiony sprawdzał stan danych na jaki oczekiwali, ponieważ powiadamianie jest ogólne – nie może być skierowane na konkretny wątek
 - w przykładzie *notify* działa zarówno na producenta jak i na konsumenta danych, stąd zawsze wątek sprawdza, czy stan danych umożliwia mu wykonanie zadania

Kolekcje i wielowątkowość

- Nowe klasy kolekcji dostarczone w Javie 1.2 w przeciwieństwie do swoich poprzedników nie były bezpieczne w środowisku wielowątkowym
- Aby zapewnić bezpieczną pracę nowych kolekcji w środowisku wielowątkowym należy obudować je odpowiednimi mechanizmami, np.:

```
List synchList  
    = Collections.synchronizedList(new ArrayList());  
Map synchMap  
    = Collections.synchronizedMap(new HashMap());
```

Kolekcje i wielowątkowość

- Od Javy 5.0 dostępne są dwie nowe klasy kolekcji z pakietu `java.util.concurrent`, które mogą bezpiecznie obsługiwać równoczesny dostęp do siebie przez wiele wątków, np. `ConcurrentLinkedQueue` i `ConcurrentHashMap`
- Uwaga! Iterator zwracany przez kolekcje nie zamraża kolekcji na czas przeglądania, a więc może nie odzwierciedlać zmian w kolekcji od momentu jego utworzenia

Nazwy wątków

- Każdy z wątków posiada własną nazwę
- Można ją nadać w momencie tworzenia obiektu poprzez użycie odpowiedniego konstruktora, np.:

```
Thread(String name)
```

```
Thread(Runnable target, String name)
```

- lub za pomocą metody:

```
public final void setName(String name)
```

Nazwy wątków

- Nazwę wątka można odczytać za pomocą metody:

```
public final String getName()
```

- Jeśli wątka nie nazwiemy w jawny sposób, to:
 - wątek główny nosi nazwę *main*
 - a kolejne: *Thread-0*, *Thread-1*, *Thread-2*, itd.

Grupy wątków

- Wątki mogą być grupowane poprzez typ *ThreadGroup*

```
ThreadGroup tg = new ThreadGroup("Grupa");  
Thread thread = new Thread(tg, runnable);
```

- Grupa wątków ułatwia zarządzanie wątkami o podobnym charakterze
 - operacje na grupie są realizowane na każdym z wątków grupy
 - poprzez grupę można obserwować liczebność aktywnych wątków

Podsumowanie

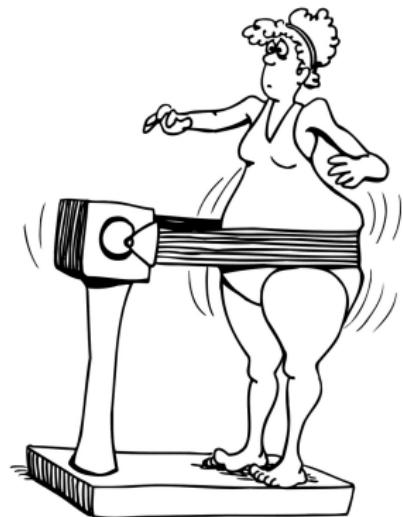
- Tworzenie aplikacji wielowątkowych
- Cykl życia wątku
- Ochrona danych
- Współpraca wątków

Pytania sprawdzające

- Czym jest wątek?
- Jakie są sposoby powołania do życia wątku?
- Czy wywołanie metody *start* powoduje, że wątek natychmiast rozpoczyna wykonywanie swojego kodu?
- Jaki efekt przyniesie wywołanie na instancji wątka metody *run* zamiast metody *start*?
- Na czym polega model wywłaszczeniowy?
- W jaki sposób wątek może samoczynnie zrzec się czasu procesora?
- Jakie właściwości mają wątki demoniczne?
- W jaki sposób można zakończyć działanie wątka?
- Jaką rolę pełni monitor obiektu i co może pełnić tę rolę?
- Do czego służy synchronizacja?
- Kiedy może wystąpić zakleszczenie?
- W jaki sposób wątki mogą ze sobą kooperować?

Ćwiczenia

- Ćwiczenie 12.1:
oczekiwanie na zakończenie wątku
- Ćwiczenie 12.2:
przykład współdziałania wątków
- Ćwiczenie 12.3:
zagadnienie producent-konsument
- Ćwiczenie 12.4:
synchronizacja wątków



Plan szkolenia

9 Zaawansowane kolekcje i we/wy

10 Biblioteka standardowa Java SE

11 Internacjonalizacja w języku Java

12 Programowanie wielowątkowe

13 Tworzenie interfejsu graficznego

14 Programowanie baz danych z użyciem JDBC API

Plan modułu

13

Tworzenie interfejsu graficznego

- założenia biblioteki graficznej
- podstawowe pojęcia przy tworzeniu GUI
- zarządzanie rozkładem komponentów
- aplikacja w Swing
- delegacyjny model zdarzeń
- budowanie menu

Założenia biblioteki graficznej

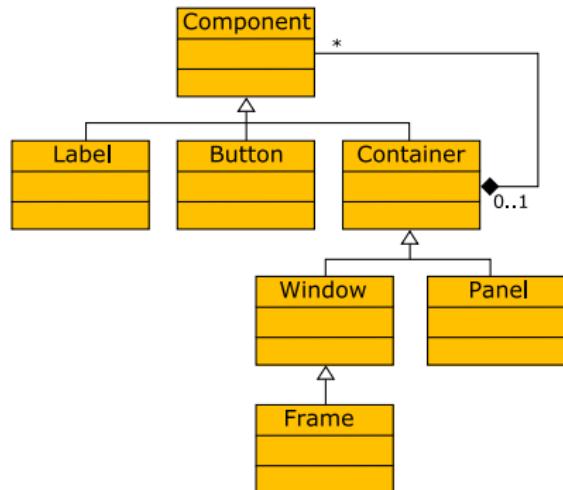
- Aplikacje Javy powinny być przenośne nie tylko na poziomie kodu źródłowego i binarnego, ale również w warstwie graficznej
- Aplikacja z GUI powinna dostosowywać się do wyglądu i funkcjonowania interfejsu graficznego danej platformy
- Biblioteka *AWT (Abstract Window Toolkit)* została przygotowana w oparciu o wybrane komponenty dostępne na platformach Windows, Motif, MacOS
- Podczas uruchomienia pod abstrakcyjne komponenty graficzne "podkładane" są komponenty dostępne na danej platformie uruchomieniowej
- Bogatszy zbiór komponentów jest dostępny w *JFC (Java Foundation Classes)* – biblioteka *Swing*

Zasady budowy interfejsu

- Budowanie graficznego interfejsu użytkownika opiera się o trzy elementy:
 - komponenty – wybór właściwych dla danego zastosowania zestawu komponentów
 - kontener – osadzenia komponentów w założonym układzie w kontenerze
 - zdarzenia – przygotowania obsługi zdarzeń związanych z poleceniami wydawanymi przez użytkownika

Podstawowe zależności między komponentami

- Dowolny obiekt typu *Component* może być dodany do obiektu typu *Container*
- Ponieważ *Component* też jest *Containerem*, to może on przyjmować również inne *Componenty*
 - dzięki tej właściwości istnieje możliwość zagnieżdżania komponentów



Komponent

- Jest to podstawowy typ, z którego są specjalizowane odpowiednie komponenty
- Zawiera kilkudziesiąt metod związanych z:
 - zarządzaniem położenia
 - obsługą zdarzeń
 - wyglądem graficznym
- Sam w sobie nie reprezentuje żadnego wyglądu i funkcjonowania
- Przy budowie GUI korzysta się z klas potomnych, np: *Button*, *Checkbox*, *Choice*, *Label*, *List*, *TextField*, *TextArea*

Kontener

- Klasa służy do agregowania innych komponentów (kontenerów)
- Posiada zestaw metod umożliwiających dodanie komponentu
- Kontener może zawierać mechanizmy sterowania rozkładem komponentów
- Podstawowe kontenery to:
 - *Panel* – kontener pomocniczy
 - *Frame*, *Window*, *Dialog* – kontenery mające swoją reprezentację graficzną

Ramka

- Ramka (*Frame*) jest podstawowym kontenerem, który może samodzielnie zaistnieć na ekranie
- Jest wyposażona w:
 - pasek tytułu
 - menu systemowe
 - przyciski sterujące rozmiarem okna
 - obwódkę, umożliwiającą płynną zmianę rozmiaru

Tworzenie ramki

- Dziedziczenie po *Frame* i uzupełnianie komponentami

```
public class MyFrame extends Frame {  
    public MyFrame() {  
        super("Tytul");  
        setSize(100, 100);  
        setVisible(true);  
    }  
}
```

- Powołanie obiektu *Frame* i zasilanie komponentami poprzez referencję

```
public class MyFrame {  
    public static void main(String a[]) {  
        Frame f = new Frame("Tytul");  
        f.setSize(100, 100);  
        f.setVisible(true);  
    }  
}
```

Zarządcy rozkładu

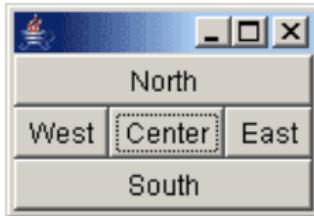
- Rozkład komponentów umieszczanych w kontenerze jest domyślnie zarządzany przez odpowiedni obiekt – **zarządecy rozkładu** (ang. *layout manager*)
 - rolą zarządcy rozkładu jest rozmieszczenie komponentów danego kontenera zgodnie z odpowiednimi regułami
 - bada preferowane rozmiary poszczególnych komponentów i na tej podstawie dobiera rozmiar kontenera
 - metoda *pack* – dynamicznie dobiera rozmiar kontenera w zależności od rozmiaru komponentów na danej platformie
 - jest odpowiedzialny za przebudowę rozkładu przy zmianie rozmiaru kontenera
- W każdym kontenerze jest domyślny zarządcy rozkładu
 - zarządcy rozkładu może być zmieniony lub wyłączony

Zarządca: BorderLayout

- Obszar kontenera jest podzielony na pięć regionów
- Do każdego regionu może być dodany jeden komponent (kontener)
 - *BorderLayout* może bezpośrednio zarządzać tylko pięcioma komponentami
- Komponenty mają rozmiar dopasowany do komórki, którą zajmują
- Przy zmianie rozmiaru kontenera
 - Obszary *North* i *South* zachowują swoją wysokość
 - Obszary *West* i *East* zachowują swoją szerokość
- *BorderLayout* jest domyślnym zarządcą dla *Frame* i *Dialog*

Zarządca: BorderLayout

```
public BLFrame() {  
    add(new Button("Center"), BorderLayout.CENTER);  
    add(new Button("South"), BorderLayout.SOUTH);  
    add(new Button("North"), BorderLayout.NORTH);  
    add(new Button("West"), BorderLayout.WEST);  
    add(new Button("East"), BorderLayout.EAST);  
    pack();  
    setVisible(true);  
}
```

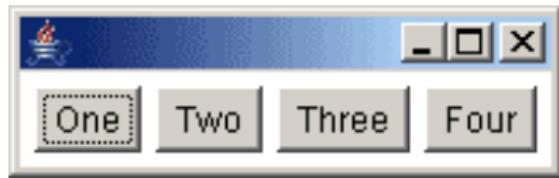


Zarządca: FlowLayout

- Kontener przyjmuje dowolną liczbę komponentów
- Komponenty są umieszczane jeden za drugim
 - gdy rozmiar okna nie pozwala na umieszczenie kolejnego komponentu w tym samym wierszu jest on umieszczany poniżej
- Komponenty są umieszczane w preferowanym przez nie rozmiarze
- *FlowLayout* jest domyślnym zarządcą dla *Panela*

Zarządca: FlowLayout

```
public FLFrame() {  
    setLayout(new FlowLayout());  
    add(new Button("One"));  
    add(new Button("Two"));  
    add(new Button("Three"));  
    add(new Button("Four"));  
    pack();  
    setVisible(true);  
}
```



Zarządca: GridLayout

- Kontener dzieli obszar na komórki ułożone w wiersze i kolumny
- Dodawane komponenty umieszczane są w kolejnych komórkach
- Rozmiar komórki dopasowany jest do największego komponentu
- Komponenty wypełniają całą przestrzeń komórki jaką zajmują

Zarządca: GridLayout

```
public GLFrame() {  
    setLayout(new GridLayout(3, 2));  
    add(new Button("One"));  
    add(new Button("Two"));  
    add(new Button("Three"));  
    add(new Button("Four"));  
    add(new Button("Five"));  
    pack();  
    setVisible(true);  
}
```

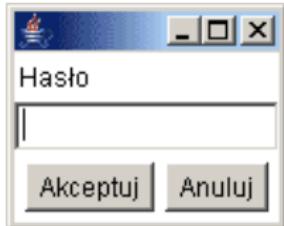


Budowanie rozkładów złożonych

- W celu realizacji rozkładu złożonego należy wykorzystać:
 - właściwości dostępnych zarządców rozkładu
 - możliwość hierarchicznego układania kontenerów
- *Panel* jest kontenerem pomocniczym przy budowaniu złożonych rozkładów
- Działanie metody *pack* wywołanej na rzecz najwyższego w hierarchii kontenera jest propagowane do każdego kontenera w nim osadzonego

Budowanie rozkładów złożonych

```
public ComplexLayout() {  
    add(new Label("Hasło"), BorderLayout.NORTH);  
    add(new TextField(), BorderLayout.CENTER);  
    Panel bottom = new Panel();  
    bottom.add(new Button("Akceptuj"));  
    bottom.add(new Button("Anuluj"));  
    add(bottom, BorderLayout.SOUTH);  
    pack();  
    setVisible(true);  
}
```



Pozycjonowanie komponentów

- Komponenty mogą być pozycjonowane bez pomocy zarządcy rozkładu
- W takim przypadku zarządcy rozkładu musi być wyłączony
 - nie należy stosować tutaj metody *pack*

```
public ManualPositioning() {  
    setLayout(null); // wyłączenie zarządcy  
    TextField tf = new TextField();  
    tf.setBounds(10, 30, 90, 30);  
    add(tf);  
    Button b = new Button("Akceptuj");  
    b.setBounds(10, 60, 90, 30);  
    add(b);  
    setSize(100, 100);  
    setVisible(true);  
}
```

- Ręcznego pozycjonowania komponentów nie należy mieszać z pozycjonowaniem kontrolowanym przez zarządcę rozkładu

Podstawowe komponenty

<i>Button</i>	przycisk, z którym może być związana akcja
<i>Checkbox</i>	pole wyboru umożliwiające zaznaczenie wybranej opcji; mogą być organizowane w grupy tworząc tzw. przyciski radiowe — możliwość zaznaczenia tylko jednego przycisku z grupy
<i>Choice</i>	opuszczane menu, umożliwiające dokonanie wyboru jednego z umieszczonych tam elementów
<i>Label</i>	element umożliwiający na umieszczenie opisu innych komponentów
<i>List</i>	lista elementów do wyboru
<i>TextField</i>	pole tekstowe umożliwiające wpisanie jednej linii tekstu
<i>TextArea</i>	pole tekstowe pozwalające na wpisywanie tekstu wierszami (notatnik)

Korzystanie z JFC (Swing)

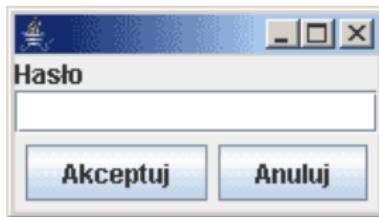
- *Swing* oferuje znacznie bogatszy zbiór komponentów
 - komponenty są rysowane od podstaw stąd istnieje możliwość zbudowania praktycznie dowolnego komponentu
- Wiele komponentów umożliwia:
 - rozdzielenie modelu danych od sposobu ich prezentacji
 - stworzenie własnych mechanizmów wprowadzania danych
- Budując interfejs oparty o *Swing* nie należy mieszać komponentów *AWT* i *JFC*
- Zobacz pakiet demonstracyjny *SwingSet* dostępny w każdej dystrybucji Javy SE

Aplikacja w Swing

- Komponenty z biblioteki Swing posiadają nazwy rozpoczynające się od litery *J*
- Dodając komponenty do kontenera głównego (*Frame*) należy skorzystać z metody zwracającej panel przeznaczony do zarządzania komponentami w imieniu ramki
- Nie mieszaj w jednej aplikacji komponentów *Swing* i *AWT*

Aplikacja w Swing

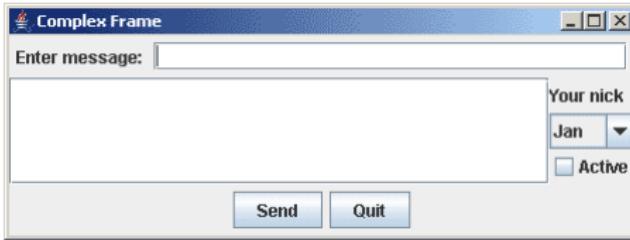
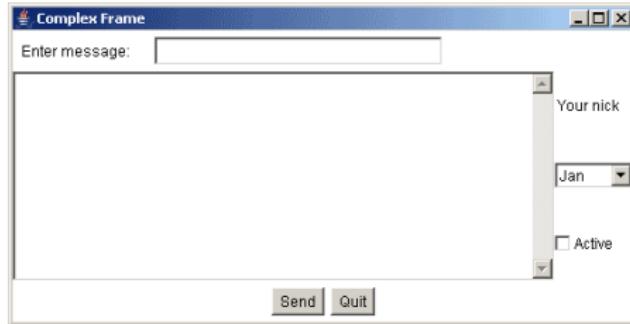
```
public class SwingDemo extends JFrame {  
    public SwingDemo() {  
        Container main = getContentPane();  
        main.add(new JLabel("Hasło"), BorderLayout.NORTH);  
        main.add(new JTextField(), BorderLayout.CENTER);  
        JPanel bottom = new JPanel();  
        bottom.add(new JButton("Akceptuj"));  
        bottom.add(new JButton("Anuluj"));  
        main.add(bottom, BorderLayout.SOUTH);  
        pack();  
        setVisible(true);  
    }  
}
```



Ćwiczenie

Ćwiczenie 13.1:

przygotowanie złożonego rozkładu – przygotuj poniższy rozkład korzystając z bibliotek AWT lub JFC



Obsługa zdarzeń

- Każdy komponent może generować różne rodzaje zdarzeń
- Obiekt, który chce odbierać zdarzenia z wybranego komponentu musi:
 - zarejestrować się w wybranym komponencie na określony typ zdarzenia
 - spełniać określone wymagania – implementować odpowiedni interfejs
- Klasa obsługująca zdarzenie musi implementować interfejs związany z danym rodzajem zdarzenia
 - np.: dla zdarzeń akcji (*Action*) – interfejs *ActionListener*
 - zdarzenia *Action* występują m.in. dla *Button*, *List*, *TextField*

Przygotowanie klasy obsługującej zdarzenia

- Obiekt będący źródłem zdarzeń rozsyła zaistniałe zdarzenie do wszystkich zarejestrowanych słuchaczy

```
ActionHandler handler = new ActionHandler();
// ...
Button b = new Button("OK");
b.addActionListener(handler);
// ...
public class ActionHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button pressed");
    }
}
```

Pozyskanie informacji o źródle zdarzenia

- Obiekt opisujący zdarzenie posiada informacje o obiekcie źródłowym jako referencję
- Zdarzenia *Action* można rozróżnić po stowarzyszonej z nimi wartości *ActionCommand*
 - domyślnie – etykieta przycisku, zawartość tekstu *TextField*
 - wartość domyślną można zmienić poprzez *setActionCommand*

```
public void actionPerformed(ActionEvent e) {  
    if (e.getSource() == button) {  
        // ...  
    } else if (e.getActionCommand().equals("OK")) {  
        // ...  
    }  
}
```

Rodzaje zdarzeń

Action	<i>ActionListener</i>	obsługa akcji wywołanych myszką lub klawiaturą
Item	<i>ItemListener</i>	oznacza zmianę wartości elementu (np. <i>Checkbox</i>)
Mouse	<i>MouseListener</i> , <i>MouseMotionListener</i>	obsługa zdarzeń powstały na skutek操作 myszką (wszystkie komponenty)
Window	<i>WindowListener</i>	obsługa operacji dotyczących okna (<i>Frame</i> , <i>Dialog</i>)
Key	<i>KeyListener</i>	obsługa klawiatury; dotyczy komponentów umożliwiających wprowadzanie tekstu

Rodzaje zdarzeń

<i>Text</i>	<i>TextListener</i>	obsługa zmian wprowadzanego tekstu (<i>TextComponent</i>)
<i>Focus</i>	<i>FocusListener</i>	przechwytywanie informacji o otrzyma- niu lub utracie sterowania przez kom- ponent
<i>Component</i>	<i>ComponentListener</i>	przechwytywanie informacji o zmia- nach komponentu
<i>Container</i>	<i>ContainerListener</i>	przechwytywanie informacji o zmia- nach w kontenerze

Zdarzenia złożone

- Wybrane zdarzenia mają więcej niż jedną metodę do obsługi, zadeklarowaną w odpowiednim interfejsie
- Obsługa zdarzeń złożonych wiąże się z implementacją wszystkich niezbędnych metod
- Klasa obsługująca zdarzenia może implementować dowolnie dużo interfejsów związanych z różnymi rodzajami zdarzeń

Zdarzenia złożone

```
public class ComplexEventHandling
    implements ActionListener, MouseMotionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println(e.getActionCommand());
    }
    public void mouseDragged(MouseEvent e) {
        System.out.println(e.getPoint());
    }
    public void mouseMoved(MouseEvent e) {
        System.out.println(e.getPoint());
    }
    public static void main(String[] args) {
        Frame f = new Frame();
        ComplexEventHandling ceh = new ComplexEventHandling();
        TextField tf = new TextField(30);
        tf.addActionListener(ceh);
        tf.addMouseMotionListener(ceh);
        f.add(tf);
        f.pack();
        f.setVisible(true);
    }
}
```

Budowanie menu

- Menu osadza się na pasku menu – *MenuBar* – który należy przywiązać do ramki
- Menu stanowi uchwyt (kontener) do osadzania elementów menu
- Element menu stanowi typ *MenuItem*
- Do elementów *MenuItem* można zarejestrować odbiorcę zdarzeń typu *Action*
- W menu można osadzać też inne elementu *Menu*, poprzez co istnieje możliwość budowania hierarchicznego menu
- W menu można osadzać elementy typu *CheckboxMenuItem* – dające możliwość zaznaczenia wyboru

Budowanie menu

```
MenuBar mb = newMenuBar();
frame.setMenuBar(mb);
Menu file = new Menu("File");
mb.add(file);
MenuItem mi = new MenuItem("Open");
file.add(mi);
Menu submenu = new Menu("Sub Menu");
file.add(submenu);
MenuItem opt = new MenuItem("Option");
submenu.add(opt);
```

Podsumowanie

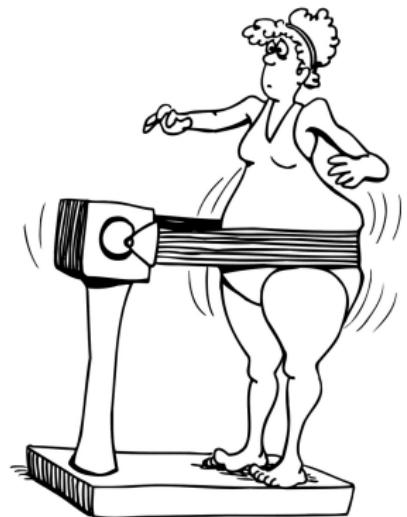
- Podstawowe pojęcia przy tworzeniu GUI
 - komponent, kontener,
 - zarządzanie rozkładem komponentów
- Delegacyjny model zdarzeń
- Tworzenie menu

Pytania sprawdzające

- Jakie podstawowe zależności zachodzą pomiędzy komponentami?
- Podaj przykłady użytecznych klas komponentów i kontenerów
- Jakie są sposoby budowy interfejsu graficznego?
- Podaj przykłady i opisz działanie zarządców rozkładu
- Jak budować rozkłady złożone?
- W jaki sposób można pozycjonować komponenty?
- Na czym polega delegacyjny model zdarzeń?
- Jakie klasy będą przydatne przy budowie menu?

Ćwiczenia

- Ćwiczenie 13.2:
*wykorzystanie delegacyjnego modelu
zdarzeń*



Plan szkolenia

9 Zaawansowane kolekcje i we/wy

10 Biblioteka standardowa Java SE

11 Internacjonalizacja w języku Java

12 Programowanie wielowątkowe

13 Tworzenie interfejsu graficznego

14 **Programowanie baz danych z użyciem JDBC API**

Plan modułu

14

Programowanie baz danych z użyciem JDBC API

- wprowadzenie do JDBC
- nawiązywanie połączenia
- przetwarzanie wyników zapytań

- Technologia umożliwiająca korzystanie z relacyjnych źródeł danych (ang. **Java DataBase Connectivity**)
- Jest to zestaw interfejsów, poprzez które aplikacja może nawiązywać połączenia, wydawać zapytania oraz przetwarzać wyniki zapytań
- Zestaw interfejsów jest implementowany poprzez sterownik realizujący protokół komunikacji z wybranym systemem relacyjnego silnika danych

DriverManager

- Sterownik *JDBC* rejestrowany jest w obiekcie *DriverManager*
 - prośba o połączenie do bazy danych jest kierowana do *DriverManager'a*
 - *DriverManager* odpytuje zarejestrowane sterowniki czy mogą obsłużyć dany rodzaj połączenia
 - jeśli zostanie znaleziony sterownik, który może obsługiwać dany rodzaj połączenia, dalsze operacje realizowane są przez implementacje interfejsów *JDBC* w tym sterowniku

Nawiązanie połączenia

- Przed nawiązaniem połączenia należy mieć zarejestrowany odpowiedni sterownik w *DriverManager*
 - do większości baz danych istnieją specjalizowane sterowniki *JDBC*
 - jeśli nie mamy sterownika do bazy, można skorzystać z mostka *ODBC* dostępnego w *Javie SE*
- Rejestracja sterownika jest realizowana przez sam sterownik w czasie jego ładowania do *JVM*, np.:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")
```

Nawiązanie połączenia

- Korzystając z *DriverManager'a* tworzymy połączenie do bazy danych podając odpowiedni URL
- URL jest zależny od sterownika bazy danych

```
String dburl = "jdbc:odbc:jpr";
Connection conn = DriverManager.getConnection(dburl);
```

Zadawanie zapytania

- *Statement* – zapytanie jednorazowe

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(
    "SELECT PASSWD FROM USERS WHERE ID='ADAM'");
```

- *CallableStatement* – wywołanie wbudowanych procedur bazy
- *PreparedStatement* – zapytanie wielokrotne, prekompilowane w trakcie tworzenia obiektu

```
PreparedStatement ps = conn.prepareStatement(
    "SELECT PASSWD FROM USERS WHERE ID=?");
ps.setString(1, "Adam");
ResultSet rs = ps.executeQuery();
```

Przeszukiwanie wyników zapytań

- Zapytania modyfikujące zwracają informacje o liczbie zaktualizowanych wpisów
- Zapytania wyszukujące zwracają wyniki w postaci obiektu *ResultSet*

```
while (rs.next()) {  
    // String passwd = rs.getString(1);  
    String passwd = rs.getString("PASSWD");  
}  
rs.close();
```

-
- Obiekt *ResultSet* nie zawiera w sobie wyników zapytania, a jedynie stanowi pośrednika pobierającego wyniki z bazy danych
 - Po zakończeniu korzystania z *ResultSet* należy go jak najszybciej zamknąć, gdyż może blokować zasoby po stronie bazy danych

Zmiany w bazie danych

- Za pomocą obiektu *ResultSet* można zmienić wartości
- Zmiana zostanie odzwierciedlona w bazie danych

```
rs.updateString("ID", "Piotr");  
rs.updateRow();
```

- W zapytaniu można określić parametry

```
PreparedStatement stmt = conn.prepareStatement(  
        "UPDATE USERS SET ID = ? WHERE ID = ?");  
stmt.setString(1, "Piotr");  
stmt.setString(2, "Adam");  
stmt.executeUpdate();
```

Pełny przykład – baza HSQLDB

```
import java.sql.*;  
  
public class DbTest {  
    public static void main(String[] args) {  
        try {  
            Class.forName("org.hsqldb.jdbcDriver");  
            Connection connection  
                = DriverManager.getConnection(  
                    "jdbc:hsqldb:file:testing.db", "sa", "");  
            Statement statement = connection.createStatement();  
            statement.executeUpdate("DROP TABLE EMPLOYEE IF EXISTS");  
            statement.executeUpdate("CREATE TABLE EMPLOYEE ("  
                + "NAME VARCHAR(255), FAMILYNAME VARCHAR(255))");  
            statement.executeUpdate("INSERT INTO EMPLOYEE "  
                + "VALUES('Jan', 'Kowalski')");  
            statement.executeUpdate("INSERT INTO EMPLOYEE "  
                + "VALUES('Arkadiusz', 'Nowak')");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Pełny przykład – baza HSQLDB

```
ResultSet resultSet
        = statement.executeQuery("SELECT * FROM EMPLOYEE");
while (resultSet.next()) {
    System.out.println("Pracownik: "
                        + resultSet.getString("NAME") + " "
                        + resultSet.getString("FAMILYNAME"));
}
statement.execute("SHUTDOWN COMPACT");
statement.close();
connection.close();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}
System.out.println("Zakonczzone");
}
```

Korzystanie z JDBC

- W przypadku aplikacji intensywnie korzystających z bazy danych należy zadbać o racjonalne zarządzanie zasobami
- Ze względu na relativnie duży koszt nawiązania połączenia do bazy danych powinny być one wykorzystywane ponownie
 - typowym rozwiązaniem jest pula połączeń, która umożliwia kontrolę liczby jednoczesnych połączeń oraz zapewnia ich dobre ponowne wykorzystanie
- Nieużywane zasoby (zapytania, wyniki) należy jak najszybciej zamykać, gdyż mogą blokować zasoby silnika bazy
- Chcąc osiągnąć wysoką efektywność komunikacji z bazą danych należy rozpatrzyć wszelkie własności stosowanego sterownika, który może przyjmować różne strategie wydawania zapytań lub pobierania wyników

Podsumowanie

- Wprowadzenie do JDBC
- Nawiązywanie połączenia
- Przetwarzanie wyników zapytań

Pytania sprawdzające

- Jaką rolę pełni *DriverManager*?
- Jak zarejestrować sterownik?
- W jaki sposób można skonstruować zapytanie do bazy danych?
- Jak odczytać wynik kwerendy?

Ćwiczenia

- Ćwiczenie 14.1:
prosta aplikacja korzystająca z JDBC

