PROGRAMOWANIE W JĘZYKU JAVA

Grupa docelowa

- Osoby pragnące rozpocząć naukę programowania w języku Java lub chcące uporządkować i poszerzyć posiadaną wiedzę

Oczekiwane przygotowanie słuchaczy

- Umiejętność posługiwania się podstawowymi narzędziami informatycznymi na poziomie średnio zaawansowanego użytkownika

MODUŁ 1 - WPROWADZENIE

Zawartość

- Historia języka
- Platforma i narzędzia
- Kompilowanie i uruchamianie kodu
- Korzystanie z dokumentacji

Historia języka Java

- Stworzony przez grupę roboczą pod kierownictwem Jamesa Goslinga z Sun Microsystems
- Opublikowany w 1995 roku jako główny składnik platformy Java
- Dostępny na zasadach licencji GNU GPL od maja 2007

Platforma Java

- Dostępna w trzech odmianach nazywanych dystrybucjami:
  a) Java Micro Edition
  b) Java Standard Edition
  c) Java Enterprise Edition
- Każda z dystrybucji występuje w dwóch wersjach:
  a) Java Runtime Environment (JRE)
  b) Java Development Kit (JDK)

Wybrane cechy języka

- Zorientowany obiektowo
- Silnie typowany
- Przenośny na poziomie źródeł i plików binarnych
- Bezpieczny i prosty w użyciu

Wirtualna Maszyna Javy

- Stanowi definicję hipotetycznego komputera – określa składnię plików klas, zestaw możliwych instrukcji, rejestrów, układ pamięci oraz inne
- Może być realizowana poprzez emulację programową lub sprzętową
- Ładuje i wykonuje kod binarny programu

Proces ładowania kodu

- Kod aplikacji ładowany jest do JVM poprzez Class Loader
- Załadowany kod jest weryfikowany przez Bytecode Verifier, a następnie tłumaczony na kod maszynowy danej platformy operacyjnej przez Translator

Class Loader

- Mechanizm ładujący kod binarny klas z sieci lub dysku do maszyny wirtualnej
- Każda z załadowanych klas umieszczana jest w odpowiedniej przestrzeni nazw co pozwala na zastosowanie odpowiednich reguł bezpieczeństwa

Bytecode Verifier

- Odpowiada za weryfikację załadowanego kodu klas
- Sprawdzane są między innymi:
  a) zgodność ze specyfikacją
  b) poprawność typów zmiennych
  c) poprawność wywołań metod
  d) poprawność konwersji danych

Translator

- Dokonuje tłumaczenia kodu pośredniego na kod maszynowy danego środowiska uruchomieniowego
- Podczas translacji wykorzystywane są mechanizmy optymalizujące takie jak:
  a) Just In Time – kompilacja w locie
  b) HotSpot – dynamiczna optymalizacja kodu

Kod źródłowy

- Ma postać zwykłych plików tekstowych z rozszerzeniem .java
- Nazwa pliku odpowiada nazwie klasy publicznej
- Pojedynczy plik może zawierać co najwyżej jedną klasę publiczną

Pierwsza aplikacja

- Plik HelloWorld.java

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

Kompilacja kodu źródłowego

- Odbywa się przy użyciu narzędzia javac wchodzącego w skład środowiska jdk
- Zasady kompilacji:
  a) jeśli istnieją źródła i pliki binarne klas zależnych używane są binaria chyba, że źródła są bardziej aktualne
  b) jeśli brakuje plików źródłowych używane są pliki binarne
  c) jeśli brakuje plików binarnych używane są pliki źródłowe
- W wyniku kompilacji powstają pliki z kodem pośrednim o rozszerzeniu .class

Przykład użycia kompilatora:
javac HelloWorld.java

Uruchamianie aplikacji

- Uruchomienie kodu aplikacji polega na wywołaniu JVM ze wskazaniem na skompilowany plik klasy zawierającej metodę main

Przykład uruchomienia aplikacji:

java HelloWorld

Dokumentacja kodu

- Komentarz jednowierszowy

Przykład:

```
// Ta linia kodu jest zakomentowana
```

- Komentarz wielowierszowy

Przykład:

```
/* To jest komentarz wielowierszowy,
rozciągający się na trzy
kolejne wiersze */
```

Javadoc

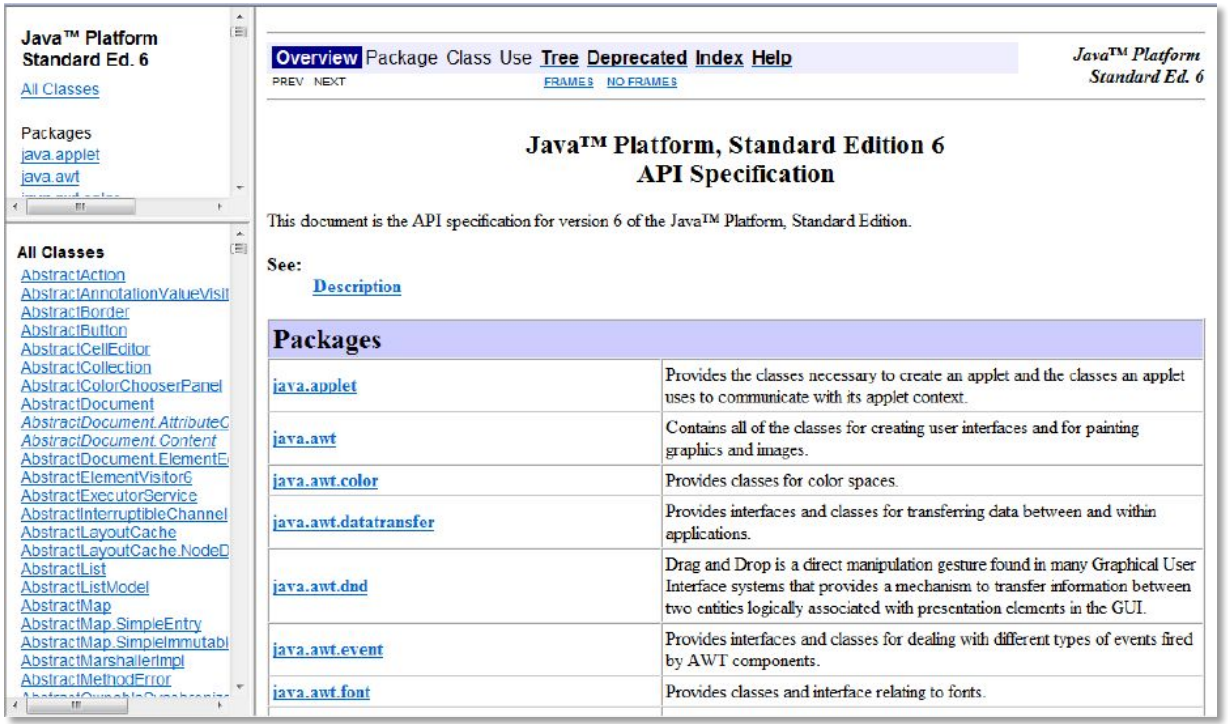
- Mechanizm pozwalający na automatyczne generowanie dokumentacji w postaci HTML
- Źródłem informacji są:
  a) definicje klas i ich składników
  b) zawartość zamknięta w ramach /** */
- Komentarz dokumentujący może zawierać dodatkowe znaczniki informacyjne np. @author, @version

Javadoc

Przykład:

```
/**
* 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 języka Java



Java™ Platform
Standard Ed. 6

All Classes

Packages
java.applet
java.awt

All Classes
AbstractAction
AbstractAnnotationValueVisit
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
AbstractDocument.AttributeC
AbstractDocument.Content
AbstractDocument.ElementE
AbstractElementVisitor6
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache
AbstractLayoutCache.NodeD
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutabl
AbstractMarshallerImpl
AbstractMethodError

Overview Package Class Use Tree Deprecated Index Help
PREV NEXT FRAMES NO FRAMES

Java™ Platform Standard Ed. 6

Java™ Platform, Standard Edition 6
API Specification

This document is the API specification for version 6 of the Java™ Platform, Standard Edition.

See:
Description

Packages	
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	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.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.

Podsumowanie

- Wymień rodzaje dystrybucji Javy ?
- Wskaż różnicę pomiędzy JRE a JDK?
- Jaką rolę pełni mechanizm Garbage Collector?
- Na jakim poziomie zagwarantowana jest przenośność kodu?
- Czym jest i za co odpowiada wirtualna maszyna Javy (JVM)?
- Jakie warunki musi spełniać plik z kodem źródłowym Javy ?
- Jakich poleceń należy użyć, aby skompilować i uruchomić program stworzony w Javie?
- W jaki sposób można utworzyć dokumentację projektu ?

MODUŁ 2 - OBIEKTY I TYPY PROSTE

Zawartość

- Typy proste
- Konwersja typów
- Typy referencyjne
- Definiowanie klas i ich składników
- Konstruktory
- Zasięg zmiennych

Typy proste

- Typy proste, nazywane podstawowymi stanowią jedyny nieobiektowy składnik języka – w ich przypadku z identyfikatorem zmiennej skojarzona jest bezpośrednio wartość, a nie wskazanie na obiekt
- Służą do reprezentowania wartości:
  a) logicznych
  b) znakowych
  c) całkowitych
  d) zmiennoprzecinkowych

Typ logiczny

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

Przykłady deklaracji:

```
boolean test;
boolean prawda = true;
```

Typ znakowy

- Nazwa typu: char
- Przechowuje kod znaku w systemie Unicode
- Zakres wartości: 16 bit [0 do 2^{16}-1]

Przykłady deklaracji:

```
char c = 'c';
char ogonki = '\u0205';
```

Typy całkowite

- Nazwy typów: byte, short, int, long
- Przechowują liczby całkowite ze znakiem
- Zakres wartości:
  a) byte – 8 bit [-128 do +127]
  b) short – 16 bit [-2^{15} do $+2^{15}-1$]
  c) int – 32 bit [-2^{31} do $2^{31}-1$]
  d) long – 64 bit [-2^{63} do $2^{63}-1$]

Przykłady deklaracji:

```
byte miesiac = 11;
int temperatura = -34;
long długość = 1285L;
int octal = 077;
int hex = 0xFF;
```

Typy zmiennoprzecinkowe

- Nazwy typów: float, double
- Przechowują wartości zmiennoprzecinkowe ze znakiem
- Zakres wartości:
  a) float - 32 bit [IEEE754]
  b) double - 64 bit [IEEE754]

Przykłady deklaracji:

```
float wynik = 10.13F;
double wartosc = 6.41e37;
```

Konwersja typów

- Automatyczna konwersja typów może zajść tylko, jeśli nie istnieje groźba utraty przechowywanej informacji

Przykłady:

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

- W pozostałych przypadkach należy zastosować rzutowanie jawne

Przykłady:

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

Typy referencyjne

- Każdy język programowania posiada określony sposób dostępu do danych
- W Javie wszystko oprócz tzw. typów prostych traktowane jest jako obiekt
- Identyfikator lub inaczej referencja pozwala na dostęp do właściwości i zachowań wskazywanego obiektu
- Jeden obiekt może być dostępny poprzez kilka referencji

Tworzenie obiektów

- Obywa się przy użyciu operatora new, którego działanie może być interpretowane w następujący sposób: „Utwórz jeden nowy obiekt danego typu”
- Operator alokuje pamięć niezbędną do przechowywania obiektu i zwraca referencję, którą można przypisać identyfikatorowi

Przykład:

```
String imie = new String("Jan");
```

Niszczenie obiektów

- Ze względów bezpieczeństwa odbywa się w sposób automatyczny - poprzez mechanizm Garbage Collector, który kontroluje ilość referencji do każdego z istniejących obiektów

Typ referencyjny String

- Nazwa typu: String
- Przechowuje łańcuchy znaków

Przykłady deklaracji:

```
String tekst = "To jest tekst";
String imie = new String("Marek");
```

- Jeśli jednym z argumentów operatora „+” jest element typu String nastąpi automatyczna konwersja i łączenie łańcuchów znaków

Przykład:

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

Typ wyliczeniowy enum

- Nazwa typu: enum
- Pozwala na tworzenie wyliczeń stałych
- Umożliwia definiowanie pól oraz metod
- Rozszerza klasę Enum

Przykład:

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

Definiowanie nowych typów

- Odbywa się przez stworzenie klasy stanowiącej szablon na bazie którego tworzone będą nowe obiekty
- Definicja klasy może zawierać:
  a) modyfikator klasy: public, <pusty>
  b) składniki klasy:
     - atrybuty
     - metody
     - konstruktory
- Składnia:

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

Atrybuty klasy

- Określają właściwości przyszłych obiektów
- Mogą być zarówno typami prostymi jak i referencyjnymi
- Mogą mieć przypisane wartości początkowe
- Składnia:

<modyfikator> <typ> <nazwa> [= <wartość>];

Metody klasy

- Określają możliwe zachowania - są operacjami, jakie może wykonywać obiekt na swoich danych
- Mogą służyć do odczytywania i zmiany stanu obiektu
- Składnia:

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

Argumenty metod

- Metody klasy mogą przyjmować listę argumentów
- Składnia:

[final? typ nazwa,]* [typ ... nazwa]?

Przykład:

public int setDate(int year, int month, int day) { … }

- Wywołanie metody z argumentami powoduje przekazanie do jej ciała ich kopii przy czym:
  a) w przypadku typów prostych jest to kopia wartości
  b) w przypadku typów referencyjnych jest to kopia referencji

Konstruktory klasy

- Specjalne metody wywoływane podczas procesu tworzenia obiektów - służą do wykonywania operacji mających na celu przygotowanie ich do pracy
- Składnia:

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

Konstruktor domyślny

- Konstruktor domyślny to inaczej konstruktor bezargumentowy
- Każda klasa musi posiadać chociaż jeden konstruktor
- Jeżeli programista nie zdefiniuje żadnego konstruktora kompilator utworzy bezargumentowy konstruktor domyślny
- Mechanizm nie zadziała jeśli w sposób jawny określony zostanie chociaż jeden konstruktor

Pakiety

- Mechanizm pozwalający na grupowanie klas o podobnym charakterze funkcjonalnym
- Każda klasa musi należeć do jakiegoś pakietu, jeśli nie zostanie on jawnie określony użyty zostanie pakiet domyślny
- Pakiety mogą zawierać w sobie inne podpakiety
- Struktura pakietowa odpowiada strukturze katalogów na dysku

Pakiety

- Każda klasa może przynależeć wyłącznie do jednego pakietu jednocześnie
- Poprzez odpowiednie modyfikatory dostępu można ograniczyć dostęp do klasy i jej składników na poziomie pakietowym
- Przynależność do pakietu jest deklarowana przy użyciu słowa kluczowego package w pierwszej linii znaczącej kodu źródłowego
- Składnia:

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

Przykład:

```
package aplikacja.gui;
```

Importowanie

- Korzystanie z klas należących do innych pakietów jest możliwe poprzez użycie pełnej nazwy pakietowej lub dokonanie importu
- Import rozszerza przestrzeń nazw o wskazany pakiet lub klasę (z wyłączeniem podpakietów)

Przykład:

```
/* rozszerzenie przestrzeni nazw o cały pakiet */
import java.util.*;
/* włączenie wskazanej klasy do przestrzeni nazw */
import java.util.Date;
```

- W przypadku wystąpienia konfliktu nazw, klasy muszą być adresowane pełną nazwą pakietową

Struktura kodu źródłowego

- Plik źródłowy może zawierać:
  a) deklarację przynależności do pakietu
  b) instrukcje importu
  c) najwyżej jedną definicję klasy publicznej
  d) definicje klas pakietowych
- Składnia:

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

Zasięg zmiennych klasowych

- Zmienne klasy (atrybuty) widoczne są dla wszystkich metod i konstruktorów klasy
- Ich czas życia jest tożsamy z czasem życia obiektu
- Nie muszą być inicjowane (choć jest to zalecane) - podczas tworzenia obiektu przypisywane są im wartości domyśle:
  a) boolean – false
  b) byte, short, int, long, char, float, double – 0
  c) typy referencyjne – null

Zasięg zmiennych metod

- Mają charakter lokalny
- Ich zasięg i czas życia jest ograniczony do czasu wykonywania metody
- Atrybuty klasy przesłonięte przez zmienne metod są dostępne przy użyciu słowa kluczowego this, będącego referencją do bieżącego obiektu

Zasięg zmiennych blokowych

- Mają charakter lokalny – są dostępne wewnątrz bloku w którym zostały zadeklarowane
- Atrybuty klasy przesłonięte przez zmienne blokowe są dostępne przy użyciu słowa kluczowego this, będącego referencją do bieżącego obiektu

Podsumowanie

- Jakie znasz nieobiektowe składniki języka Java ?
- W jakim celu stosuje się rzutowanie ?
- Opisz różnicę między typem prostym i referencyjnym
- W jaki sposób tworzy się nowe obiekty ?
- Wymień dozwolone składniki klasy
- Co to jest i do czego służy konstruktor klasy ?
- Czym są pakiety ?
- W jaki sposób przekazywane są argumenty metod ?

MODUŁ 3 - PODSTAWY SKŁADNI JĘZYKA

Zawartość

- Identyfikatory
- Operatory
- Instrukcje sterujące
- Tablice

Identyfikator

- Identyfikator to inaczej nazwa klasy, metody lub zmiennej
- Może składać się z dowolnej liczby znaków alfanumerycznych przy czym:
  a) pierwszy znak musi być literą, znakiem "_" lub "$"
  b) nazwa nie może być tożsama z żadnym ze słów kluczowych
  c) rozróżniane są wielkie i małe litery
  d) mogą występować znaki narodowe (nie zalecane) – litery są reprezentowane w systemie Unicode

Przykłady:

value, param1, błąd, thisMan, _param, $money$

Konwencja kodowania

- Dokument stworzony przez firmę Sun Microsystems, dostępny pod adresem http://java.sun.com/docs/codeconv/
- Opisuje zalecaną konwencję formatowania kodu Javy
- Pozwala na zwiększenie czytelności plików źródłowych i stanowi ogólnie przyjęty standard wśród programistów

Konwencja nazewnicza klas

- Nazwy klas powinny przyjmować formę rzeczownika i rozpoczynać się od wielkiej litery
- W przypadku kiedy nazwa składa się z wielu członów, każdy z nich piszemy łącznie i rozpoczynamy od wielkiej litery

Przykłady:

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

Konwencja nazewnicza metod

- Nazwy metod powinny przyjmować formę czasownika i rozpoczynać się od małej litery
- W przypadku kiedy nazwa składa się z wielu członów, każdy z nich piszemy łącznie i rozpoczynamy od wielkiej litery

Przykłady:

assignTripToCustomer, withdrawMoney

Konwencja JavaBeans

- Nazwy metod dostępowych zgodnych z konwencją JavaBeans powinny rozpoczynać się od odpowiedniego przedrostka ("set", "get" lub "is") po którym następuje nazwa atrybutu pisana wielką literą
- W przypadku kiedy nazwa składa się z wielu członów, każdy z nich piszemy łącznie i rozpoczynamy od wielkiej litery
- Klasy typu JavaBeans powinny posiadać bezargumentowy konstruktor

Przykłady:

setName, setPhoneNumber, setEndFlag

getBalance, getMaxNumber, isAdmin

Konwencja nazewnicza atrybutów

- Nazwy zmiennych metod i atrybutów klas powinny przyjmować formę rzeczownika i rozpoczynać się od małej litery
- W przypadku kiedy nazwa składa się z wielu członów, każdy z nich piszemy łącznie i rozpoczynamy od wielkiej litery
- W drodze wyjątku dopuszczalne jest stosowanie jednoliterowych nazw jako liczników pętli
- Nazwy stałych powinny być pisane wielkimi literami, a ewentualne człony rozdzielone znakami podkreślenia

Przykłady:

counter, currentTime, bigValue

MAX_VALUE, TIME_LIMIT

Formatowanie kodu źródłowego

- Długość wiersza nie powinna przekraczać 80 znaków
- Łamanie linii może się odbywać po przecinku, przed operatorem i jeśli to możliwe należy zachowywać wyrażenia w całości (np. wywołania funkcji)
- Należy stosować wcięcia kodu, aby odzwierciedlić logikę algorytmu
- Dopuszczalne jest wprowadzenie lokalnych zmiennych pomocniczych w celu uproszczenia kodu
- Wszystkie stosowane identyfikatory powinny być jasne i samoopisujące

Operatory

- Pozwalają na wykonywanie operacji:
  a) arytmetycznych
  b) logicznych
  c) bitowych
  d) porównania
  e) przypisania

Operatory arytmetyczne

- Możliwe operacje arytmetyczne:
  a) mnożenie: x = x * y;
  b) dzielenie: x = x / y;
  c) reszta z dzielenia całkowitego: x = x % y;
  d) dodawanie: x = x + y;
  e) odejmowanie: x = x - y;
  f) inkrementacja zmiennej: ++x; lub x++;
  g) dekrementacja zmiennej: −−x; lub x−−;
  h) nadanie wartości x wyniku operacji z y: x [op]= y; jest równoważne z: x = x [op] y; gdzie [op] jest jednym z operatorów: +, -, *, /, %

Operatory logiczne

- Możliwe operacje logiczne:
  a) negacja: !x
  b) operacja AND: x && y
  c) operacja OR: x || y
  d) wyrażenia logiczne zawierające && lub || ulegają skracaniu np.:

  `if (x != null && x.isValid()) {}` w przypadku, gdy x ma wartość null, drugie wyrażenie nie będzie rozwinięte, ponieważ można już określić wartość końcową całego wyrażenia – fałsz

  `if (x == null || x.isValid()) {}` w przypadku, gdy x ma wartość null, drugie wyrażenie nie będzie wykonane, ponieważ można już określić wartość końcową całego wyrażenia - prawda

Operatory bitowe

- Możliwe operacje bitowe:
  a) operacja AND: x & y 01001011 & 00001111 = 00001011
  b) operacja OR: x | y 01100011 | 11000000 = 11100011
  c) operacja XOR: x ^ y 00110011 ^ 01100110 = 01010101
  d) przesunięcie w lewo: x << y
  00101010 << 1 = 01010100 równoważne z 32 << 1 = [32 * 2] = 64
  e) przesunięcie w prawo z uzupełnieniem bitami znaku: x >> y
  00101010 >> 1 = 00010101 równoważne z 32 >> 1 = [32 / 2] = 16
  f) przesunięcie w prawo z uzupełnieniem zerami: x >>> y
  101010 >>> 1 = 010101
  a) negacja logiczna: ~x ~101010 = 010101

Operatory relacyjne

- Możliwe operacje relacyjne:
  a) większy: x > y
  b) większy lub równy: x >= y
  c) mniejszy: x < y
  d) mniejszy lub równy: x <= y
  e) równy: x == y
  f) różny: x != y

Kolejność operatorów

- Kolejność wykonywania operatorów
  a) () . []
  b) (rzutowanie) ! ~ ++ --
  c) * / %
  d) + -
  e) << >> >>>
  f) < <= > >=
  g) == !=
  h) &
  i) ^
  j) |
  k) &&
  l) ||
  m) ? :
  n) = += -= *= %= itd.

Instrukcja warunkowa if

- Składnia:

```
if (wyrażenie logiczne) {
  // operacje
}
```

lub

```
if (wyrażenie logiczne) {
  // operacje wykonywane, jeśli wyrażenie logiczne jest prawdziwe
} else {
  // operacje wykonywane, jeżeli wyrażenie logiczne jest fałszywe
}
```

Pętla for

- Składnia:

```
for (wyrażenie początkowe; warunek logiczny; wyrażenie pętli) {
  // blok operacji
}
```

Przykład:

```
for (int i = 0; i < 10; i++)
  System.out.println(i);
}
```

Nowa pętla for

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 (wyrażenie logiczne) {
  // blok operacji
}
```

Przykład:

```
int i = 11;
while (i > 0) {
  System.out.println(i--);
}
```

Pętla do…while

Składnia:

```
do {
  // blok operacji
} while (wyrażenie logiczne);
```

Przykład:

```
int i = 11;
do {
  System.out.println(i--);
} while(i > 0);
```

Instrukcje break/continue

- Instrukcja break przerywa wykonanie aktualnej pętli, continue powoduje zakończenie bieżącej iteracji
- Etykiety pozwalają określić pętlę docelową, ich użycie jest opcjonalne

Przykład:

```
int i = 500;
while (i > 10) {
  // operacje
  if (i % 100 == 0) {
    continue;
  }
  i--;
}
```

Instrukcja switch

- Argumentem instrukcji switch musi być wartość typu: int, short, byte, char lub enum
- Składnia:

```
switch (wartość całkowita) {
  case wartość1:
    operacje;
    break;
  case wartość2:
    operacje;
    break;
  default:
    operacje;
}
```

Tablice

- Pozwalają na grupowanie zmiennych jednakowego typu
- Każda tablica jest obiektem dlatego musi zostać stworzona przy użyciu operatora new
- Rozmiar tablicy jest ustalany podczas jej tworzenia i nie może zostać zmieniony w późniejszym czasie
- Indeks tablicy zmienia się od 0 do n-1 gdzie n to liczba elementów
- Kontrola indeksów tablicy jest prowadzona przez JVM

Przykłady:

```
int[] tab;
double[] tab2 = new double[10];
int tab2[] = tab; // powielona jest tylko referencja
```

Tablice

- Tablica typów prostych przechowuje ich wartości, natomiast tablica typów referencyjnych wskazania na obiekty

Przykłady:

```
Date daty[] = new Date[3];
daty[0] = new Date(1999, 10, 12);
daty[1] = new Date(1000, 10, 10);
daty[2] = new String("kot"); // źle, próba umieszczenia referencji innego typu
```

Tablice

- Istnieje możliwość wstępnej inicjalizacji tablicy, w takim przypadku rozmiar wyznaczany jest na podstawie ilości podanych elementów

Przykłady:

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

Przeglądanie elementów tablicy

Przykłady:

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

lub

```
for (int tmp : tab) {
  System.out.println(tmp);
}
```

Tablice wielowymiarowe

- Tablica dwuwymiarowa to inaczej tablica tablic
- W podobny sposób można tworzyć tablice o większej liczbie wymiarów
- Każdy z wymiarów posiada własny atrybut length

Przykłady:

```
char [][] tab;
char []tab[];
int tab[][] = new int[5][10];
Date d[][] = new Date[2][]; (ang. jagged array)
d[0] = new Date[6];
d[1] = new Date[8];
```

Tablice wielowymiarowe

Przykład:

```
String msg [][] = {{"ala", "ola", "ela"}, {"kot", "pies"}};
for (int i = 0; i < msg.length; i++) {
  for (int j = 0; j < msg[i].length; j++) {
    System.out.println(msg[i][j]);
  }
}
```

Podsumowanie

- Jakie warunki spełnia poprawny identyfikator w Javie ?
- Podaj zasady konwencji nazewniczej dla klas, metod i atrybutów
- Jaka jest konstrukcja instrukcji warunkowej ?
- Jak można wymusić zakończenie pętli ?
- Opisz składnię instrukcji switch
- Jak uzyskać rozmiar tablicy ?
- Jak numerowane są elementy tablicy ?
- Jak można skopiować zawartość jednej tablicy do innej ?
- Jak utworzyć tablicę dwuwymiarową w której każdy wiersz będzie miał inną liczbę elementów ?

MODUŁ 4 - PROGRAMOWANIE OBIEKTOWE

Zawartość

- Podstawowe pojęcia obiektowe
- Relacje między obiektami
- Dziedziczenie i polimorfizm
- Interfejsy i klasy abstrakcyjne
- Klasy wewnętrzne

Programowanie obiektowe

- Paradygmat programowania, w którym program zdefiniowany jest jako zbiór współpracujących i komunikujących się wzajemnie obiektów, których celem jest realizacja założonego zadania

Klasy i obiekty

- Klasa stanowi ogólną reprezentację danego typu
- Obiekt jest indywidualnym przedstawicielem utworzonym na podstawie definicji klasy
- Klasa stanowi "formę" z której wytwarzane są obiekty o identycznych własnościach i zachowaniach jednak mogące różnić się stanem

Podstawowe pojęcia obiektowe

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

Abstrakcja danych

- Abstrakcja danych to inaczej generalizacja modelu rzeczywistości
- Pozwala na modelowanie złożonych zagadnień i systemów na takim poziomie ogólności, który pozwala na rozwiązanie problemu
- W miarę potrzeb model może być coraz bardziej uszczegóławiany

Hermetyzacja danych

- Hermetyzacja (ang. encapsulation) oznacza ukrywanie danych, a tym samym wewnętrznego i delikatnego stanu obiektu przed światem zewnętrznym

Przykład bez hermetyzacji:

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

Date d = new Date();
d.month = 2;
d.day = 33;
```

Hermetyzacja danych

Przykład z hermetyzacją danych:

```
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 dostępny
d.setDay(32);  // wartość nie zostanie zaakceptowana
```

Hermetyzacja danych

Inny przykład z hermetyzacją danych:

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

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

Relacje między obiektami

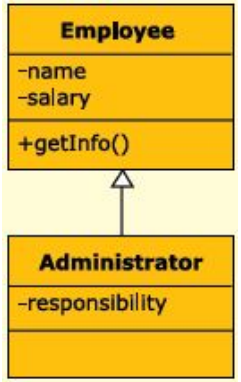
- Określają rolę pełnioną przez obiekt będący w związku
- Asocjacja – występuje kiedy jeden obiekt korzysta z usług drugiego, oba współpracują, aby zrealizować wspólny cel np. Driver-Car
- Agregacja – stanowi rozwinięcie asocjacji, występuje kiedy jeden obiekt uzupełnia się lub zawiera kolekcje innych obiektów np. Group-User
- Kompozycja – najsilniejsza ze wszystkich relacji, występuje kiedy jeden obiekt zawiera w sobie lub składa się z innych obiektów, jednocześnie ich samoistne istnienie nie ma większego sensu np. Clock-Hand

Dziedziczenie

- Mechanizm pozwalający na tworzenie nowych typów na bazie już istniejących - nowa klasa potomna dziedziczy cechy klasy bazowej
- Jest relacją generalizacji-specjalizacji
- Java pozwala na dziedziczenie tylko z jednej klasy jednocześnie

Przykład:

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



Dziedziczenie i konstruktor

- Konstruktor klasy bazowej nie jest dziedziczony
- Klasa potomna oczekuje, że klasa bazowa będzie posiadała konstruktor domyślny i jeśli taki nie istnieje należy jawnie wywołać inny, właściwy konstruktor
- Wywołanie musi nastąpić w pierwszej linii znaczącej konstruktora klasy potomnej przez użycie słówka super

Dziedziczenie i konstruktor

Przykład:

```
public class Animal {
  String name;
  public Animal(String name) {
    this.name = name;
  }
}
public class Dog extends Animal {
  public Dog(String name) {
    super(name); // wywołanie 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

Przedefiniowywanie metod

- Klasa potomna może dostarczyć nowej definicji dla metody dziedziczonej z klasy bazowej (ang. override)
- Nadpisywana metoda musi spełniać następujące warunki:
  a) musi posiadać taką samą nazwę
  b) musi posiadać taką samą listę argumentów
  c) musi zwracać wartość tego samego typu lub typu pochodnego
  d) nie może oferować szerszego poziomu dostępu
- Dostęp do oryginalnej metody z klasy bazowej można uzyskać poprzez słowo kluczowe super

Przedefiniowywanie metod

Przykład:

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

Polimorfizm

- Polimorfizm, wielopostaciowość – oznacza, że dana referencja może wskazywać na wiele różnych form spokrewnionych obiektów przez co ich zachowanie będzie wydawało się różne
- Referencja typu bazowego nie pozwala na dostęp do specyficznych cech typu potomnego

Przykład:

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

Uogólnione argumenty metod

Przykład:

```
public String prepare (Vehicle v) {
  v.start();
}

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

Kolekcje heterogeniczne

Przykład:

```
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();
}
```

Konwersja typów

- Referencja typu bazowego może wskazywać na obiekty potomne, w tym wypadku konwersja typu zachodzi automatycznie

Przykład:

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

- Referencja typu potomnego może potencjalnie wskazywać na obiekt typu bazowego jednak wymaga to jawnej konwersji typów oraz ich zgodności

Przykład:

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

- Konwersja typów z dwóch sąsiadujących gałęzi hierarchii nie jest możliwa

Konwersja typów

- Sprawdzanie poprawności konwersji zachodzi w czasie wykonywania
- W celu zwiększenia bezpieczeństwa należy zbadać możliwość jej przeprowadzenia - operator instanceof

Przykład:

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

Proces inicjacji obiektu

- Tworzenie obiektu przy użyciu operatora new przebiega w następujący sposób:
  a) alokacja pamięci potrzebnej do stworzenia obiektu
  b) nadanie wartości domyślnych zmiennym klasy
  c) wywołanie konstruktora
  d) podstawienie wartości pod argumenty konstruktora
  e) wywołanie konstruktora klasy bazowej
  f) dalej inicjacja obiektu bazowego zachodzi w analogiczny sposób aż do poziomu obiektu Object
  g) inicjacja zmiennych klasy wartościami określonymi przez programistę
  h) wykonanie instrukcji konstruktora
  i) zwrócenie referencji do obiektu

Proces inicjacji obiektu

- Konstruktor nie powinien wywoływać metod, które mogą być przedefiniowane w klasie potomnej, ponieważ może dojść do wywołania polimorficznego z poziomu konstruktora i próby użycia zmiennych, które nie zostały jeszcze zainicjowane
- Jeśli konstruktor wywołuje metody powinny być one oznaczone jako prywatne

Klasa Object

- Klasa stojąca na szczycie hierarchii obiektów w Javie
- Definiuje metody dziedziczone przez wszystkie klasy czyli:
  a) toString() – zwraca reprezentację tekstową obiektu
  b) equals(Object o) – służy do porównywania obiektów
  c) hashCode() – zwraca w miarę unikatową wartość typu int zależną od stanu obiektu

Porównywanie obiektów

- Przy użyciu operatora porównania "==" można sprawdzić czy referencja wskazuje dokładnie na ten sam obiekt
- Metoda equals() dziedziczona z Object działa identycznie, po nadpisaniu może porównywać stany obiektów np. equals z klasy String porównuje, czy łańcuchy obu obiektów są tej samej długości i czy składają się z tych samych znaków
- Nowo tworzone klasy powinny nadpisywać metodę equals oraz hashCode

Metoda equals()

- Zwraca prawdę jeśli porównywany obiekt jest równy obiektowi na rzecz którego wywoływana jest metoda
- Dla każdego obiektu x różnego od null x.equals(x) powinno zwracać true
- Dla wszystkich x i y różnych od null x.equals(y) powinno zwracać true jeśli y.equals(x) zwraca true
- Dla wszystkich x, y, z różnych od null jeśli x.equals(z) zwraca true i y.equals(z) zwraca true to x.equals(y) zwraca true
- Dla każdego x różnego od null x.equals(null) zawsze zwraca false

Metoda hashCode()

- Zwraca wartość kodu mieszającego dla obiektu (mapuje stan obiektu na jednolity skalar typu int)
- Każde wywołanie metody powinno zwrócić tą samą wartość (jeśli stan obiektu nie uległ w międzyczasie zmianie)
- Jeśli dwa obiekty są równe według metody equals() to oba muszą mieć te same wartości kodu mieszającego
- Nie jest wymagane, aby dla dwóch obiektów, które są różne według metody equals(), wartości kodów mieszających również były różne. Jest to natomiast wskazane ze względu na wydajność mechanizmów opartych o kody mieszające

Przeciążanie metod

- Ma miejsce kiedy dwie lub więcej metod klasy posiadają tą samą nazwę, ale różnią się listą przyjmowanych argumentów
- Różnica wynikająca tylko z typu wartości zwracanej i modyfikatora dostępu jest niewystarczająca, ponieważ kompilator musi rozpoznać, które ciało metody wstawić podczas wywołania

Przykład:

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

Przeciążanie konstruktorów

- W każdej klasie można zdefiniować dowolną liczbę konstruktorów różniących się liczbą i rodzajem przyjmowanych argumentów
- Wywołanie konstruktora z konstruktora jest możliwe dzięki słowu kluczowemu this i musi nastąpić jako pierwsza linia znacząca konstruktora

Przykłady:

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

Elementy statyczne

- Elementy związane z klasą, a nie konkretnym obiektem – są współdzielone przez wszystkie instancje, można się do nich odwołać bez konieczności tworzenia obiektu
- Mogą:
  a) służyć do przechowywania danych wspólnych dla całej rodziny obiektów
  b) kontrolować liczbę instancji danego typu
  c) udostępniać metody, które nie muszą operować na danych obiektu
- Do definicji elementów statycznych służy modyfikator static
- Nie ma możliwości odwołania się do elementów nie statycznych z wnętrza metod statycznych

Elementy statyczne

Przykład:

```
public class Test {
  static int a;
  int b;
  public static void go() { // ... }
  public void show() { // ... }
}
// elementy statyczne przez nazwę klasy
Test.a = 10;
Test.go();
// elementy obiektu tylko przez referencję
Test test = new Test();
test.b = 10;
test.show();
```

Blok statyczny

- Wykonywany przed utworzeniem jakiegokolwiek obiektu danego typu
- Może być stosowany do przeprowadzenia dodatkowej inicjalizacji
- W bloku statycznym nie można korzystać z obiektów typu, w którym jest osadzony

Przykład:

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

Elementy finalne

- Znaczenie słowa kluczowego final zależy od kontekstu użycia:
  a) w przypadku klasy oznacza, że nie może być ona rozszerzana
  b) w przypadku metody, że nie można jej przedefiniować
  c) w przypadku zmiennej, że nie można zmienić jej wartości
- 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

Klasy abstrakcyjne

- To klasy w których nie wszystkie metody zostały zaimplementowane
- Mogą służyć do stworzenia ogólnej klasy bazowej (szablonu) na poziomie której nie można podać rozsądnej implementacji metod dziedziczonych przez określoną rodzinę obiektów
- Do oznaczenia klasy abstrakcyjnej służy słowo abstract
- Na podstawie klas abstrakcyjnych nie można tworzyć obiektów

Klasy abstrakcyjne

Przykład:

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

Klasy abstrakcyjne

- Klasa posiadająca przynajmniej jedną metodę abstrakcyjną musi być oznaczona jako abstrakcyjna
- Klasa dziedzicząca po klasie abstrakcyjnej musi zaimplementować wszystkie metody abstrakcyjne, w przeciwnym przypadku sama musi być oznaczona jako abstrakcyjna
- Klasa abstrakcyjna posiada konstruktor mimo iż nie można powołać obiektu na podstawie takiej klasy – jest on wykorzystywany podczas procesu tworzenia obiektów klas potomnych
- Klasa abstrakcyjna może posiadać elementy statyczne oraz blok statyczny

Interfejsy

- Interfejsy to wbudowany mechanizm języka pozwalający na zmniejszenie sprzężenia między elementami systemu
- Definicja interfejsu odbywa się przez użycie słowa kluczowego interface
- Interfejs może być rozpatrywany jako specjalny przypadek klasy abstrakcyjnej, w której wszystkie metody są abstrakcyjne
- Interfejs nie może zawierać deklaracji atrybutów chyba że są one oznaczone jako finalne
- Wszystkie metody interfejsu są publiczne i finalne (nawet jeśli jawnie nie zostanie to określone)

Interfejsy

- Pozwalają na symulację dziedziczenia wielobazowego – każda klasa może implementować zachowanie zadeklarowane w kilku interfejsach
- Mogą pełnić rolę znacznikową
- Interfejs może rozszerzać dowolną liczbę innych interfejsów
- Klasa może implementować dowolną liczbę interfejsów
- Klasa implementująca określony interfejs jest instancją jego typu
- Klasa implementująca interfejs zobowiązana jest do zaimplementowania wszystkich jego metod; w przeciwnym przypadku musi być oznaczona jako abstrakcyjna

Interfejsy

Przykład:

```
public interface Printable {
  public void printInfo();
}
public class Report implements Printable {
  public void generateReport() {
    ...
  }
  public void printInfo() {
    ...
  }
}
```

Klasy wewnętrzne

- To klasy definiowane wewnątrz innych klas
- Mają dostęp do wszystkich składników klasy otaczającej (także prywatnych)
- Mogą być oznaczone jako prywatne i tym samem niewidoczne dla świata zewnętrznego
- Ich instancja jest zawsze powoływana w kontekście klasy zewnętrznej

Klasy wewnętrzne

Przykład:

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

Jakość modelu obiektowego

- Zwartość lub inaczej spójność (ang. cohesion) jest miarą jak bardzo klasa lub grupa klas przyczynia się do realizacji określonego celu
- Należy dążyć do jak największej spójności

Jakość modelu obiektowego

- Sprzężenie (ang. coupling) – jest miarą określającą wzajemną zależność klas
- Należy dążyć do jak najmniejszego sprzężenia - obiekty powinny być od siebie zależne tylko w zakresie niezbędnym do realizacji założonego zadania

Podsumowanie

- Na czym polega polimorfizm ?
- Jak można przedefiniować metodę w klasie potomnej ?
- Czy poprzez referencję typu bazowego można wywołać metodę klasy potomnej ?
- Za pomocą jakiego operatora można sprawdzić, czy referencja jest danego typu ?
- Jak przebiega proces tworzenia nowego obiektu ?
- Podaj przykłady metod dziedziczonych z klasy Object ?
- W jaki sposób sprawdzić, czy dwa różne obiekty są identyczne ?
- Jakie cechy maja elementy statyczne klasy ?
- Jakie konsekwencje pociąga za sobą użycie słowa kluczowego final ?
- Do czego mogą być przydatne interfejsy ?

MODUŁ 5 - OBSŁUGA BŁĘDÓW I WYJĄTKÓW

Zawartość

- Rodzaje sytuacji wyjątkowych
- Obsługa wyjątków
- Tworzenie własnych typów wyjątków

Czym są sytuacje wyjątkowe

- Do sytuacji wyjątkowych należą:
  a) błędy wynikające ze środowiska uruchomieniowego
     - brak dostępnej pamięci operacyjnej
     - błąd biblioteki systemowej lub JVM
  b) błędy spowodowane przez otoczenie aplikacji
     - niedostępność urządzeń wejścia/wyjścia
     - przerwane połączenia sieciowe
  c) błędy spowodowane przez programistę
     - korzystanie z niezainicjowanych zmiennych
     - przekroczenie indeksów tablicy
     - dzielenie przez zero

Obsługa sytuacji wyjątkowych

- Każda sytuacja wyjątkowa złapana przez JVM opisana jest odpowiednim obiektem wywodzącym się z typu Throwable
- Błędy wynikające ze środowiska uruchomieniowego pochodzą z rodziny Error, nie muszą i wręcz nie powinny być obsługiwane przez programistę
- Błędy powodowane przez otoczenie aplikacji pochodzą z rodziny Exception i muszą być obowiązkowo obsłużone – należy dostarczyć kod wywoływany w momencie wystąpienia wyjątku
- Błędy spowodowane przez programistę pochodzą z rodziny RuntimeException nie muszą być obsługiwane, ponieważ dobrze napisany kod ich nie generuje

Blok chroniony

- Służy do wykonywania instrukcji potencjalnie niebezpiecznych
- Po bloku chronionym umieszcza się bloki obsługi wyjątków wykonywane w przypadku wystąpienia sytuacji wyjątkowej

Przykład:

```
try {
  // operacje, które potencjalnie niebezpieczne
} catch (IOException e) {
  // kod obsługi wyjątku
} catch (AWTException e) {
  // kod obsługi wyjątku
}
```

Blok chroniony – kolejność wykonania instrukcji

Przykład:

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

Kolejność przechwytywania

- Obsługa wyjątku zostaje przekazana do pierwszego, pasującego bloku catch dlatego należy zacząć od wyjątków najbardziej szczegółowych

Przykład:

```
try {
  …
} catch(FileNotFoundException e) {
  // obsługa
} catch (IOException e) {
  // obsługa
} catch (AccessDeniedException e) {
  // obsługa
}
```

Obsługiwać czy przechwytywać

- Miejsce powstania wyjątku nie zawsze jest odpowiednie do jego obsługi. W takim wypadku można go przekazać do metody wywołującej, która będzie zobowiązana na niego zareagować
- Proces przekazywania wyjątku może odbywać się wielokrotnie
- Metoda wykonująca operacje potencjalnie niebezpieczne i nie zapewniająca ich obsługi musi zadeklarować listę wyrzucanych wyjątków przy użyciu słowa throws

Przedefiniowywanie metod

- Istnieje możliwość przedefiniowywania metod wyrzucających wyjątki, jeśli spełnione są następujące warunki:
  a) nowa metoda nie rozszerza listy wyjątków typu „checked”
  b) nowa metoda nie będzie generować wyjątków bardziej ogólnych - ze względu na zachowanie polimorficzne, klient metody bazowej nie może być "zaskoczony" nagłym pojawieniem się wyjątku
  c) przedefiniowana metoda może zmniejszyć listę wyjątków lub je całkowicie zlikwidować
  d) przedefiniowana metoda może rozszerzyć listę wyjątków o wyjątki typu „runtime”

Blok finally

- Gwarantuje wykonanie zbioru instrukcji niezależnie od tego czy sytuacja wyjątkowa miała miejsce czy też nie

Przykład:

```
try {
  openConnection();
   …
   return x;
} catch (ConnectionException e) {
  // obsługa
} finally {
  closeConnection();
}
```

Własne typy wyjątków

- Powstają poprzez rozszerzenie jednej z istniejących klas wyjątków, najczęściej klasy Exception
- Do wyrzucenia wyjątku służy słowo kluczowe throw

Przykład:

```
public class MyException extends Exception {
  public MyException() {}
  public MyException(String s) { super(s); }
}
void met1() throws MyException {
  if (wrong) throw new MyException();
  …
}
```

Podsumowanie

- Jakie znasz kategorie sytuacji wyjątkowych ?
- Które z wyjątków należy obsługiwać ?
- Jaką składnię ma blok chroniony?
- Co to jest stos wywołań ?
- Kiedy obsługę wyjątku należy przekazać do metody nadrzędnej ?
- Jak stworzyć własny typ wyjątku ?

MODUŁ 6 - ŚRODOWISKO SYSTEMOWE

Zawartość

- Korzystanie z klas systemowych
- Tworzenie dystrybucji aplikacji
- Kolekcje danych
- Klasy opakowujące
- Formatowanie wyjścia
- Adnotacje

Sterowanie aplikacją

- Podczas uruchamiania aplikacji można przekazać parametry z linii poleceń, zostaną one podstawione jako argumenty metody main
- Badając długość tablicy można zweryfikować ich ilość

Przykład:

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

Zmienne systemowe

- Środowisko, w którym działa aplikacja dostarcza aplikacji w postaci zmiennych systemowych
- Dostęp do zmiennych środowiskowych jest możliwy przy użyciu metody System.getenv(String);

Przykłady:

```
java -Dmoja.zmienna=wartość
```

```
Properties p = System.getProperties();
p.list(System.out);
String os = p.getProperty("moja.zmienna");
```

Dystrybucja aplikacji

- To nic innego jak archiwum grupujące wszystkie klasy i zasoby wchodzące w skład aplikacji
- W najprostszym wypadku można je utworzyć w następujący sposób:

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

Dystrybucja aplikacji

- Istnieje możliwość utworzenia archiwum pozwalającego na bezpośrednie uruchamianie aplikacji
- W tym celu należy przygotować specjalny plik MANIFEST.MF, który będzie zawierał informacje o klasie posiadającej metodę main

Przykład:

Wpis do pliku MANIFEST.MF

Main-Class: pl.BizTech.MojaKlasa

Utworzenie archiwum

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

Uruchomienie aplikacji

java -jar <nazwa_archiwum>

Klasa StringBuffer

- Reprezentuje rozszerzalny bufor znaków i pozwala na przechowywanie oraz modyfikację łańcuchów znaków bez każdorazowej konieczności tworzenia nowego obiektu
- Klasa udostępnia wiele metod pozwalających na przeprowadzanie operacji na przechowywanym tekście między innymi:
  a) zamianę znaków
  b) wstawianie znaków na wybranej pozycji
  c) wycinanie fragmentów łańcucha

Przykład:

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

Klasa StringBuilder

- Klasa oferuje identyczną funkcjonalność jak omówiony wcześniej StringBuffer jednak różni się pod względem oferowanej wydajności – ze względu na to, że nie zapewnia synchronizacji i bezpieczeństwa w aplikacjach wielowątkowych jest znacznie wydajniejsza

Klasa Locale

- Reprezentuje informacje na temat bieżącej lokalizacji
- Może być wykorzystana przez klasy prezentujące dane użytkownikowi w celu dopasowania ich formy do wariantu językowego/lokalizacyjnego

Przykład:

```
Locale polska = new Locale("pl", "PL");
NumberFormat nf = NumberFormat.getCurrencyInstance(polska);
double pensja = 3533.12;
System.out.println(nf.format(pensja));
```

Formatowanie wyjścia

- Java udostępnia możliwość formatowania wyjścia przy użyciu metod:
  a) printf(String format, Object ... args);
  b) format(String format, Object ... args);
- Do oznaczania miejsc, w których mają być wstawione argumenty oraz ich typów służą specjalne znaczniki:
  a) %b – wartość typu boolean lub null
  b) %c – znak w standardzie Unicode
  c) %d – liczba dziesiętna
  d) %f – liczba zmiennoprzecinkowa
  e) %s – ciąg znaków

Przykład użycia:

```
System.out.printf("U˙zytkownik \%s ma ID \%d", "Nowak", 1);
```

Klasa NumberFormat

- Dostarcza interfejs pozwalający na formatowanie danych liczbowych w zależności od bieżącej lokalizacji
- Lokalizacja może dotyczyć między innymi:
  a) getNumberInstance – zwykłych liczb
  b) getCurrencyInstance – walut
  c) getPercentInstance – procentów

Przykład:

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

Klasa DateFormat

- Dostarcza interfejs pozwalający na formatowanie daty i czasu w zależności od bieżącej lokalizacji
  a) getDateInstance – pobiera format daty
  b) getTimeInstance – pobiera format czasu
  c) getDateTimeInstance – pobiera format daty i czasu
- Wszystkie powyższe metody fabryki mogą przyjmować jako argument styl wyświetlania oraz lokalizację
- Dostępne style:
  a) DateFormat.SHORT – 29.12.05
  b) DateFormat.MEDIUM – 2005-12-29
  c) DateFormat.LONG – 29 grudzień 2005
  d) DateFormat.FULL – czwartek, 29 grudzień 2005

Klasa SimpleDateFormat

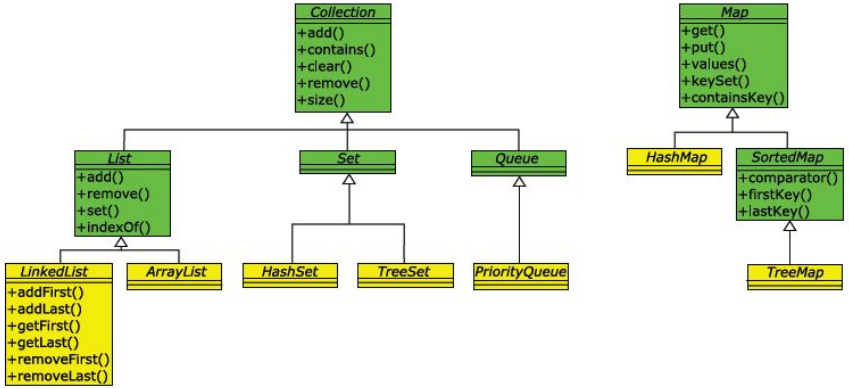
- Pozwala na zdefiniowanie niestandardowego wzorca daty lub czasu i przeprowadzanie konwersji data<->text

Przykład:

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

Collection API

- Kolekcje to inaczej kontenery pozwalające na grupowanie i przechowywanie zbioru obiektów
- Każdy typ kolekcji oferuje inne właściwości np.:
  a) Set – gwarantuje unikatowość elementów
  b) List – pozwala na uporządkowanie składników
  c) Map –



Tworzenie kolekcji

Przykład:

```
Collection<Object> set = new HashSet<Object>();
set.add(new String("jan"));
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();
```

Kolekcje generyczne

- Kolekcje generyczne, inaczej typowane, wymuszają kontrolę typu elementów przeprowadzaną w czasie kompilacji
- Pozwalają na zwiększenie bezpieczeństwa oraz wyeliminowanie rzutowania

Przykład:

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

Przeglądanie kolekcji - iterator

Przykład:

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

Przeglądanie kolekcji - pętla for

Przykład:

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

Sortowanie kolekcji - interfejs Comparable<T>

- Interfejs Comparable pozwala na ustalenie naturalnego uporządkowania elementów kolekcji
- Określenie kolejności odbywa się przy użyciu metody `public int compareTo(Object obj);` lub `public int compareTo(T obj);` w przypadku generycznej wersji interfejsu

Sortowanie kolekcji - interfejs Comparable

Przykład:

```
public class Person implements Comparable {
  private String name;
  private String secondName;
  @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);
  }
}
```

Sortowanie kolekcji - interfejs Comparable<T>

Przykład:

```
public class Person implements Comparable<Person> {
  private String name;
  private String secondName;
  @Override
  public int compareTo(Person person) {
    int secondNameComparison
        = secondName.compareTo(person.secondName);
    if (secondNameComparison != 0) {
      return secondNameComparison;
    }
    return name.compareTo(person.name);
  }
}
```

Sortowanie kolekcji - interfejs Comparable<T>

Przykład:

```
Person person1 = new Person("Jan", "Kowalski");
Person person2 = new Person("Piotr", "Kami´nski");
Person person3 = new Person("Aleksander", "Wielki");
List<Person> persons = new ArrayList<Person>();
persons.add(person1);
persons.add(person2);
persons.add(person3);
Collections.sort(persons);
```

Sortowanie kolekcji - interfejs Comparator<T>

- Jeśli zachodzi potrzeba sortowania według różnych kryteriów można skorzystać z interfejsu Comparator definiuje on metodę: `int compare(Object o1, Object o2);` lub `int compare(T o1, T o2);` w wersji generycznej

Sortowanie kolekcji - interfejs Comparator<T>

Przykłady:

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

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

Sortowanie kolekcji - interfejs Comparator<T>

Przykład:

```
Person person1 = new Person("Jan", "Kowalski");
Person person2 = new Person("Piotr", "Kami´nski");
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());
```

Klasy opakowujące

- Każdy z typów podstawowych posiada swój odpowiednik obiektowy tzw. typ opakowujący np.:
  a) char - Character
  b) int – Integer
  c) short - Short
  d) long - Long
  e) double – Doube
  f) float – Float
  g) boolean – Boolean
- Wszystkie obiekty opakowujące są niemutowalne
- Wraz z Javą w wersji 5.0 pojawił się mechanizm automatycznego boxingu/unboxingu (pakowania/wypakowywania) elementów typu prostego

Klasy opakowujące

Przykład:

```
Collection<Integer> set = new ArrayList<Integer>();
Collection<int> set2 = new ArrayList<int>(); // źle!!!
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

- Mechanizm pozwalający na umieszczanie dodatkowych meta-informacji na poziomie kodu źródłowego
- Standardowe adnotacje Javy to:
  a) @Override
  b) @Deprecated
  c) @SuppressWarnings

Przykład:

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

Podsumowanie

- Jak można przekazać parametry startowe do aplikacji ?
- W jaki sposób można utworzyć dystrybucję aplikacji ?
- Wymień klasy operujące na łańcuchach znaków i scharakteryzuj czym się różnią ?
- Opisz krótko poznane typy kolekcji
- Jakie zastosowanie mają typy generyczne ?
- W jaki sposób można zrealizować przeglądanie kolekcji ?
- Do czego służą klasy opakowujące ?
- Podaj przykłady użycia adnotacji

MODUŁ 7 - OPERACJE WEJŚCIA/WYJŚCIA

Zawartość

- Obsługa operacji wejścia/wyjścia

Strumienie

- Mechanizm pozwalający na uszeregowane przesyłanie danych (tekstowych lub binarnych) między dwoma węzłami
- Węzłami mogą być:
  a) aplikacje
  b) pliki
  c) gniazdka sieciowe
  d) wątki
- Strumienie są zawsze jednokierunkowe
- Pojedynczy strumień posiada wejście (input) i wyjście (output)

Strumienie binarne i tekstowe

- Strumienie binarne InputStream oraz OutputStream służą do przesyłania surowych danych binarnych, które mogą być następnie poddane interpretacji
- Strumienie tekstowe Reader i Writer pozwalają na przesyłanie danych tekstowych, zapewniają odpowiednią konwersję strony kodowej znaków

Binarny strumień wejściowy

- InputStream – pozwala na sekwencyjny odczyt bajtów
- Udostępnia metody:
  a) read() – zwraca odczytany bajt
  b) read(byte[] buff) – wczytuje dane do podanej tablicy
  c) read(byte[] buff, int off, int len) – wczytuje dane do podanej tablicy, począwszy od off, w ilości len
  d) markSupported() – sprawdza, czy strumień może być znacznikowany
  e) mark(int readLimit) – ustawia znacznik
  f) reset() – powrót do znacznika
  g) close() – zamyka strumień

Binarny strumienie wyjściowy

- OutputStream – pozwala na sekwencyjny zapis bajtów
- Udostępnia metody:
  a) write(int b) – zapisuje bajt danych do strumienia
  b) write(byte[] buff) – zapisuje dane z podanej tablicy
  c) write(byte[] buff, int off, int len) – zapisuje dane z podanej tablicy, począwszy od off, w ilości len
  d) flush() – "wypłukuje" dane ze strumienia
  e) close() – zamyka strumień

Tekstowy strumienie wejściowy

- Reader – pozwala na sekwencyjny odczyt znaków
- Udostępnia metody:
  a) read() – zwraca odczytany znak
  b) read(char[] buff) – wczytuje dane do podanej tablicy
  c) read(char[] buff, int off, int len) – wczytuje dane do podanej tablicy, począwszy od off, w ilości len
  d) markSupported() – sprawdza, czy strumień może być znacznikowany
  e) mark(int readAheadLimit) – ustawia znacznik
  f) reset() – powrót do znacznika
  g) close() – zamyka strumień

Tekstowy strumienie wyjściowy

- Writer – pozwala na sekwencyjny zapis znaków
- Udostępnia metody:
  a) write(int c) – zapisuje znak do strumienia
  b) write(char[] buff) – zapisuje dane z podanej tablicy
  c) write(char[] buff, int off, int len) – zapisuje dane z podanej tablicy, począwszy od off, w ilości len
  d) write(String s) – zapisuje dane z podanego obiekty typu String
  e) write(String s, int off, int len) – zapisuje dane z podanego obiektu String, począwszy od off, w ilości len
  f) flush() – "wypłukuje" dane ze strumienia
  g) close() – zamyka strumień (zapewnia przesłanie ich do odbiorcy)

Operacje na plikach

- Współpraca z plikami wymaga otwarcia odpowiedniego strumienia
- W zależności od tego, czy chcemy pracować z plikiem binarnym, czy tekstowym wybieramy właściwą parę strumieni:
  a) FileInputStream, FileOutputStream – pliki binarne
  b) FileReader, FileWriter – pliki tekstowe

Przykład:

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

- W celu poprawienia efektywności można zastosować filtr buforujący

Przykład:

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

Operacje na plikach

- Klasa File udostępnia metody zwracające informacje oraz pozwalające na wykonywanie podstawowych operacji na systemie plików np.: canRead, canWrite, exists, getPath, isDirectory, list, mkdir, delete, rename

Przykład:

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

Filtrowanie danych

- Strumienie stanowią wygodną formę przesyłu danych, ale można je także wykorzystać do przeprowadzania transformacji
- W tym celu na strumień należy nałożyć jeden lub więcej filtrów (dekoratorów)
- Najczęstsze zastosowania:
  a) buforowanie danych
  b) przekształcanie danych do odpowiednich typów
  c) kompresja lub szyfrowanie
- Przykładami filtrów mogą być:
  a) DataInputStream - przekształcenie surowych danych binarnych w typy podstawowe
  b) DataOutputStream - przekształcenie typów podstawowych w dane surowe

Serializacja danych

- Serializacja polega na zamianie istniejącego obiektu na strumień bajtów i przekazanie go do odpowiedniego strumienia
- Proces deserializacji jest procesem odwrotnym
- Mechanizm może zostać wykorzystany np.: do utrwalania obiektów lub przesyłania ich przez sieć

Przykłady:

```
ObjectOutputStream oos = new ObjectOutputStream(
new FileOutputStream("user.ser"));
oos.writeObject(new Date());
oos.close();
ObjectInputStream ois = new ObjectInputStream(
new FileInputStream("user.ser"));
Date date = (Date) ois.readObject();
ois.close();
```

Serializacja danych

- Serializacji podlega cała złożona struktura obiektu (nawet jeśli składa się on z innych obiektów)
- Koniecznym warunkiem poprawnego procesu jest implementacja interfejsu znacznikowego Serializable
- Właściwości, które nie powinny być serializowane muszą zostać oznaczone modyfikatorem transient
- W klasie implementującej interfejs Serializable powinno się zadeklarować pole serialVersionUID, które umożliwia wersjonowanie klas

Przykład:

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

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
- Scanner dzieli dane źródłowe na pojedyncze tokeny poprzez odnajdywanie znaków separatora
- Separator można ustawić używając metody useDelimiter
- Do pobierania danych wejściowych udostępnia metody: next, nextInt, nextFloat, nextByte, nextBoolean itd.

Klasa Scanner

Przykład:

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

Tokenizer

- Klasa wspierająca analizę ciągów tekstowych - umożliwia podział tekstu według zadanego tokenu

Przykład:

```
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

- Do czego służą strumienie ?
- Na czym polega proces serializacji?
- Jakie wymagania muszą być spełnione, aby serializacja przebiegła w sposób prawidłowy ?
- Jakie zastosowanie ma klasa Scanner ?

MODUŁ 8 - PROGRAMOWANIE WIELOWĄTKOWE

Zawartość

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

Czym jest wątek ?

- Przepływ sekwencji sterowania w programie, posiadający własny stos wywołań
- Wszystkie wątki wykonywane w ramach jednego procesu mają i dostęp do tej samej przestrzeni adresowej
- Każda aplikacja rozpoczyna swoje działanie poprzez wątek główny (ang. main thread), który może stworzyć i rozpocząć wykonywanie innych wątków
- W rzeczywistym środowisku wątki konkurują między sobą o dostęp do fizycznych zasobów komputera dając iluzję pracy równoległej

Tworzenie wątku

- W celu stworzenia wątku należy:
  a) stworzyć klasę implementującą interfejs Runnable - instrukcje umieszczone w ramach metody run(), wynikającej z implementacji interfejsu, będą wykonywane w ramach pracy wątku
  b) przygotować nową instancję obiektu Thread podając jako argument konstruktora instancję Runnable
  c) uruchomić wątek przy użyciu metody start()

Tworzenie wątku

Przykład:

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

Tworzenie wątku inaczej

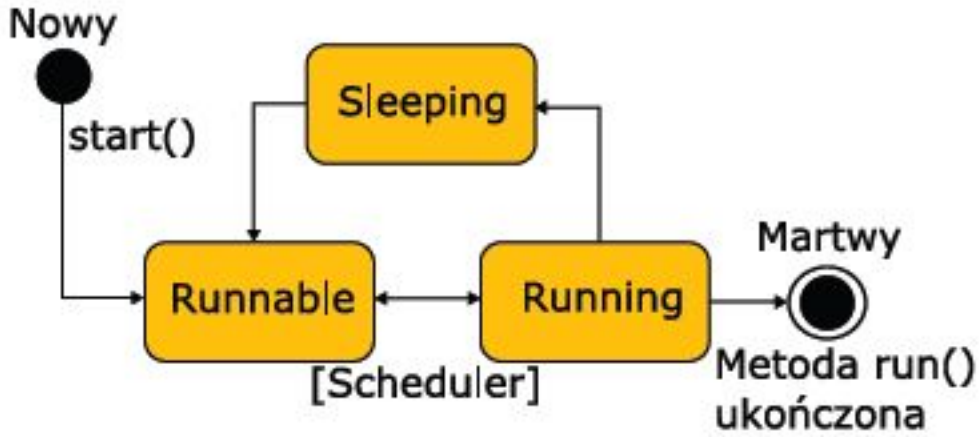
- Wątek można także utworzyć przez rozszerzenie klasy Therad i nadpisanie metody run()
- Dziedziczenie po Thread daje nieco prostszy kod, ale jednocześnie ogranicza – nie pozwala na rozszerzenie innej klasy
- Wykorzystanie interfejsu Runnable umożliwia:
  a) lepsze zastosowanie podejścia obiektowego
  b) osiągnięcie większej spójności i elastyczności

Tworzenie wątku inaczej

Przykład:

```
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();
    t1.start();
  }
}
```

Wykonywanie wątku



Wykonywanie wątków

- Model wywłaszczeniowy zakłada, że wątek jest w stanie Running, dopóki mechanizm szeregowania wątków nie zdecyduje o tym, że inny wątek powinien uzyskać dostęp do zasobów
- Java udostępnia metody statyczne pozwalające na sterowanie wykonaniem wątków:
  a) yield() – aktualny wątek zrzeka się zasobów, Scheduler wybiera z puli wątków typu Runnable inny wątek, który zajmie jego miejsce
  b) sleep() – uśpienie aktualnego wątku na określony czas

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() – powoduje zatrzymanie wykonania bieżącego wątku do czasu zakończenia pracy innego wątku lub upływu określonego okresu czasu

Przykład:

```
Thread t = new Thread(testThread);
t.join(100);
```

- Metoda setPriority() – pozwala na ustalenie priorytetu wątku
- Metoda setDaemon() – powoduje przejście wątku do stanu demonicznego
- Metoda isAlive() – zwraca informację czy wykonano metodę start() i czy wątek nie zakończył jeszcze swojego działania
- Metoda stop() – nagłe zatrzymanie pracy wątku - niezalecane

Kończenie pracy wątku

Przykład:

```
public class Test implements Runnable {
  private boolean end = false;
  public void run() {
    while (!end) {
    …
  }
  public void stopThread() {
    end = true;
  }
}
```

Kończenie pracy wątku

- Wątek może zostać bezpiecznie zakończony przy użyciu metody interrupt()

Przykład:

```
public class Test implements Runnable {
  public static void main(String[] args) throws InterruptedException {
    Thread t = new MyThread();
    t.start();
    Thread.sleep(100);
    t.interrupt();
  }
  public void run() {
    while (!Thread.currentThread().isInterrupted())
      System.out.println("Hello");
  }
}
```

Kończenie pracy wątku

Przykład:

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

Potrzeba ochrony danych

- Ponieważ wątki operują na współdzielonych danych może dojść do sytuacji w której podczas ich wywłaszczenia dane zostaną w niespójnej postaci
- Zarządzanie ochroną danych jest jednym z trudniejszych aspektów programowania wielowątkowego

Integralność danch

Przykład:

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

Ochrona danych

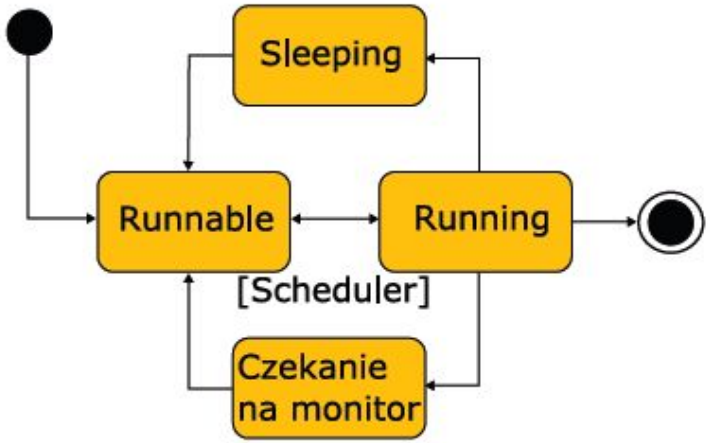
- W celu ochrony integralności danych można wykorzystać wbudowany mechanizm monitora do obiektu
- Monitor to znacznik, który w danej chwili może być przejęty tylko przez jeden wątek (poprzez realizację bloku lub metody synchronizowanej)
- Jeśli wątek posiadający monitor zostanie wywłaszczony nie oddaje on monitora
- Należy ograniczać liczbę instrukcji synchronizowanych - nie są one wykonywane współbieżnie, dlatego mogą stanowić "wąskie gardło" aplikacji. Dodatkowo powstają narzuty związane z przejęciem/zwrotem monitora

Ochrona danych

Przykład:

```
public class Storage {
  int idx;
  char tab[];
  synchronized void push(char c) {
    tab[idx] = c;
    idx++;
  }
  char pop() {
    synchronized (this) {
      idx--;
      return tab[idx];
    }
  }
}
```

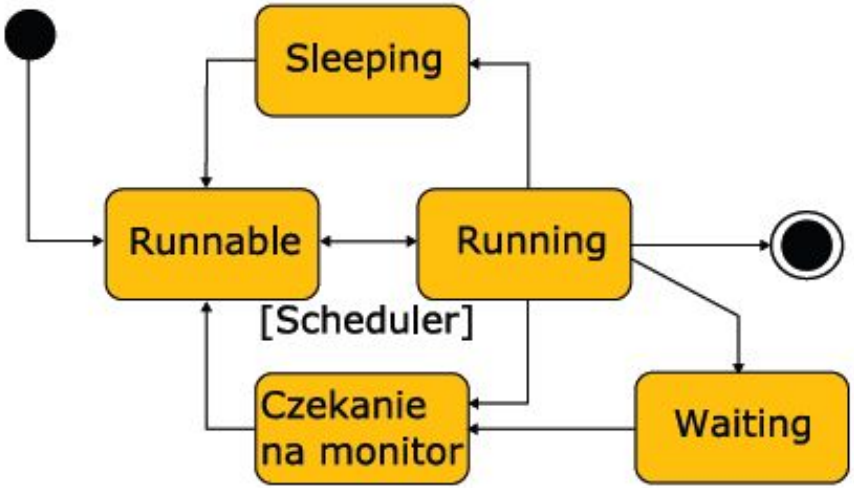
Synchronizacja wątków



Zakleszczenie

- Występuje gdy wątek posiadający monitor do obiektu oczekuje dostępu do obiektu, którego monitor jest zajęty przez inny wątek, który oczekuje na dostęp do monitora posiadanego przez wątek pierwszy
- Sytuacja zakleszczenia często jest trudna do wykrycia, ponieważ może występować tylko przy pewnej zależności czasowej
- Poszukując potencjalnego miejsca zakleszczenia należy rozpatrzeć wszystkie wywołania wymagające pobrania więcej niż jednego monitora

Współpraca wątków



Współpraca wątków

Przykład:

```
synchronized void push(char c) {
  while (indx == LIMIT) {
    try {
      wait();
    } catch (InterruptedException e) { //... }
  }
  notify();
  tab[idx] = c;
  idx++;
}
…
```

Współpraca wątków

```
synchronized char pop() {
  while (idx == 0) {
    try {
      wait();
    } catch (InterruptedException e) { //... }
  }
  notify();
  idx--;
  return tab[idx];
}
```

Nazwy wątków

- Każdy z wątków posiada własną nazwę domyślną nadawaną podczas procesu tworzenia
- Można ją zmienić poprzez użycie odpowiedniego konstruktora np.:
  a) Thread(String name)
  b) Thread(Runnable target, String name)

  lub metody:
  c) public final void setName(String name)
- Odczytanie nazwy jest możliwe przez metodę:
  a) public final String getName()

Grupy wątków

- Wątki mogą być grupowane przy użyciu typu ThreadGroup
- Grupa wątków ułatwia zarządzanie wątkami o podobnym charakterze – operacje są realizowane na każdym z wątków z osobna

Przykład:

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

Podsumowanie

- Czym jest i do czego służy wątek ?
- Jak stworzyć i uruchomić wątek ?
- Czy wywołanie metody start powoduje, że wątek natychmiast rozpoczyna wykonywanie pracę ?
- W jaki sposób wątek może zrzec się dostępu do procesora ?
- Jakie właściwości mają wątki demoniczne ?
- W jaki sposób można zakończyć działanie wątku ?
- Jaką rolę pełni monitor do obiektu ?
- Kiedy może wystąpić zakleszczenie ?
- W jaki sposób wątki mogą ze sobą kooperować ?