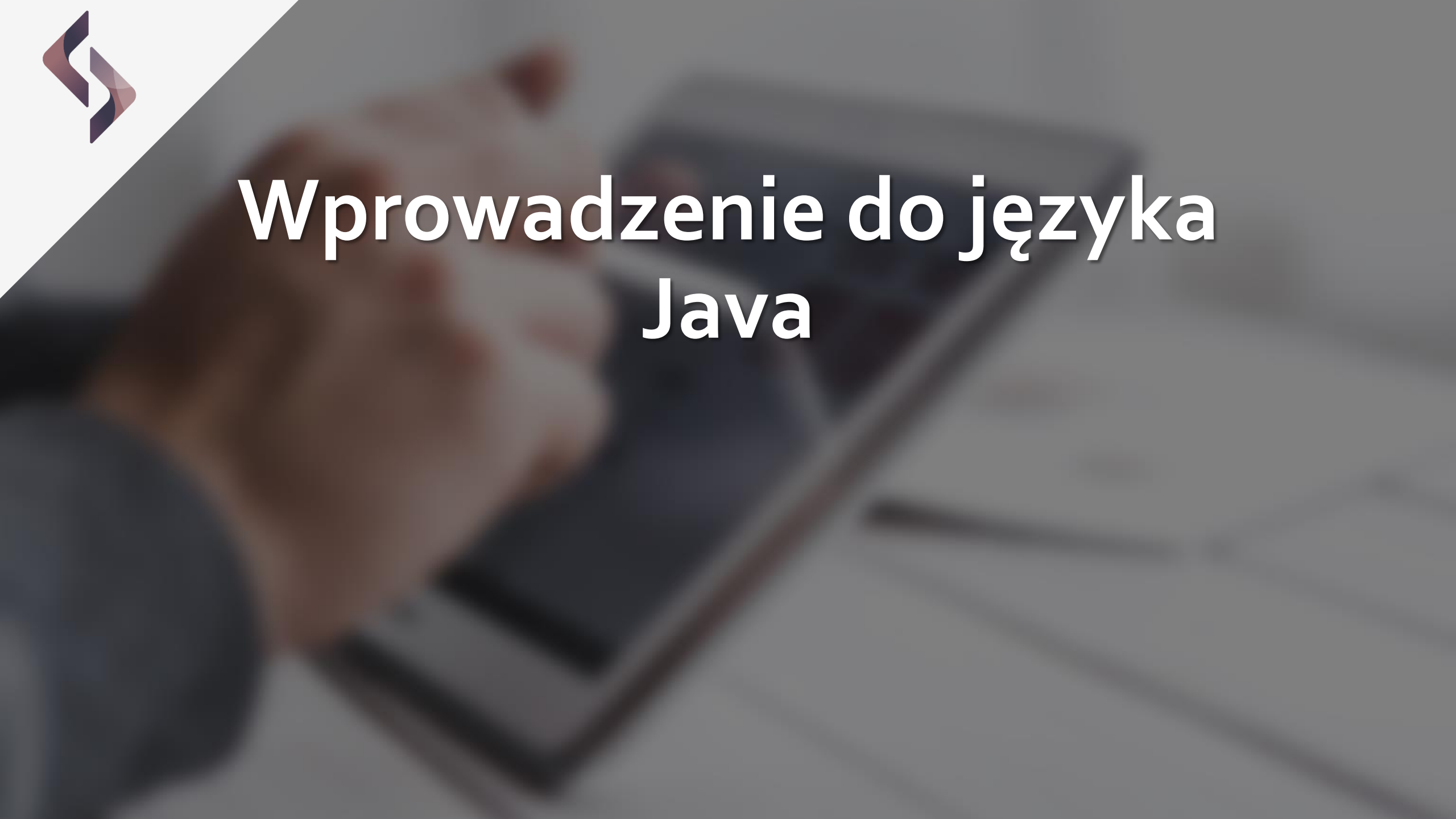
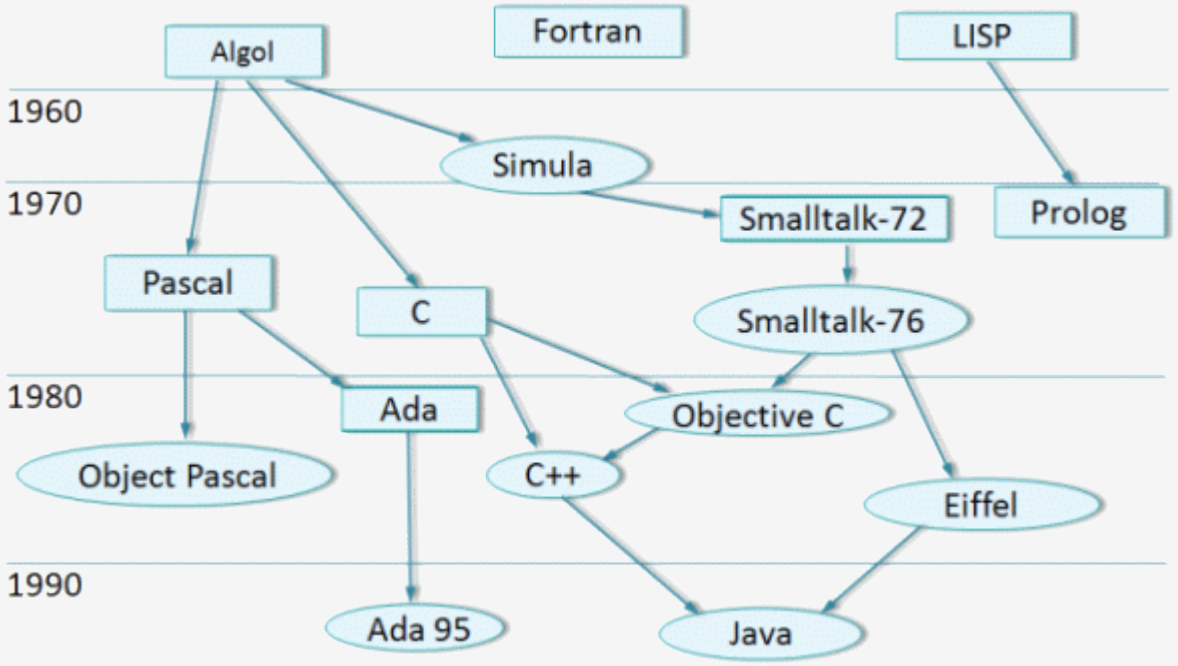


Wprowadzenie do języka Java

Historia programowania



Wraz z upływem czasu, komputery stawały się coraz szybsze oraz coraz tańsze. Karty perforowane po prostu przestały wystarczać. Pojawiły się urządzenia wejścia (klawiatury) i wyjścia (monitory, drukarki), a przede wszystkim, umożliwiono przechowywanie danych (pierwsze pamięci masowe). Dzięki temu możliwe stało się nie tylko wykonywanie, ale również przygotowanie programów na samych komputerach.

O języku Java

- Obiektowy język programowania
- Istnieje od 1995, stworzony przez Jamesa Goslinga z firmy Sun Microsystems
- Oparty na podstawie koncepcji języka Smalltalk i C++
- Obecnie możemy używać Javy w wersji 8 (stabilna) i 9 (early access)
- Dziś Java jest wspierana przez firmę Oracle

- Aplikacje uruchamiane są na maszynie wirtualnej JVM
- Zapewnia to niezależność od platformy. Wiele platform języka (Java Standard Edition, Java Enterprise Edition, Java ME, Java FX, Android SDK)
- Środowisko uruchomieniowe (JRE) Java jest zainstalowane na większości urządzeń PC.
- Początkowo zaprojektowana do tworzenia aplikacji przeglądarkowych (aplety), obecnie ma szerokie zastosowanie

Historia języka Java

Początek języka **Java** możemy określić jako rok 1991. Wtedy firma Sun z Patrickiem Naughtonem oraz Jamesem Goslingiem na czele postanowili stworzyć prosty i niewielki język, który mógłby być uruchamiany na wielu platformach z różnymi parametrami. Projekt zatytułowano **Green**.


Historia języka Java

Pierwsza wersja Javy ukazała się w 1996 roku w wersji 1.0. Niestety nie osiągnęła ona wielkiego rozgłosu z czego inżynierowie firmy Sun dokładnie zdawali sobie sprawę. Na szczęście dosyć szybko poprawiono błędy i uzupełniono ją o nowe biblioteki, model zdarzeń GUI.

Kolejne edycje Javy, to przede wszystkim dodawanie nowych funkcjonalności oraz prace nad wydajnością bibliotek standardowych. Największe zmiany zaszły chyba w wersji 5.0, gdzie wprowadzono **Klasy generyczne (Generic Classes),** typy enum, autoboxing, varargs, adnotacje.

Historia wersji języka JAVA

JDK Beta	1994	
JDK 1.0	1996	Wersja inicjalna
JDK 1.1	1997	Klasy wewnętrzne, JDBC, RMI, refleksja, JIT, AWT
J2SE 1.2	1998	Swing, kolekcje
J2SE 1.3	2000	JNDI, debugger
J2SE 1.4	2002	Nowa obsługa operacji Input/Output, wyrażenia regularne, parser XML, logowanie, exception chaining
J2SE 5.0	2005	Typy generyczne, typy wyliczeniowe, autoboxing, varargs, pętla foreach, importy statyczne, zmiany w wielowątkowości
Java SE 6	2006	Zakończenie obsługi Windows 9x, zmiany w Swingu, usprawnienie JDBC, JAX-WS, zmiany w JVM
Java SE 7	2011	Zmiana obsługi plików, zmiany w języku (switch, operator diamond, catch wielu wyjątków). Nowości w wielowątkowości.
Java SE 8	2014	Wyrażenia lambda, strumienie, nowe api dla dat i czasu. Zakończenie obsługi Windows XP

Na rok 2017 zapowiedziana została opóźniona data wydania stabilnej wersji Javy 9.

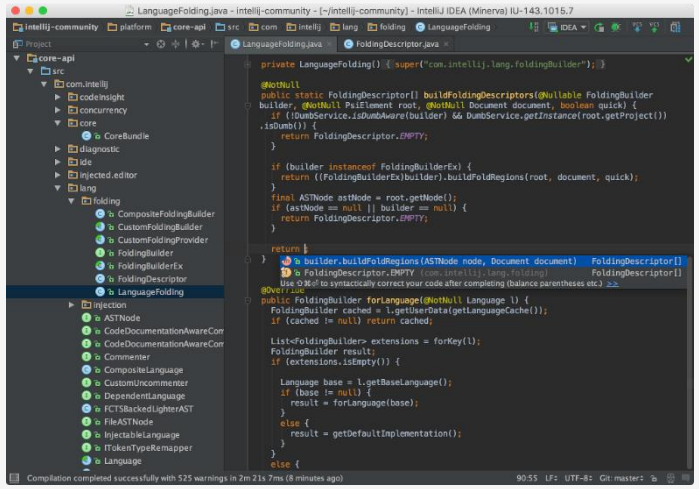
Dlaczego Java?

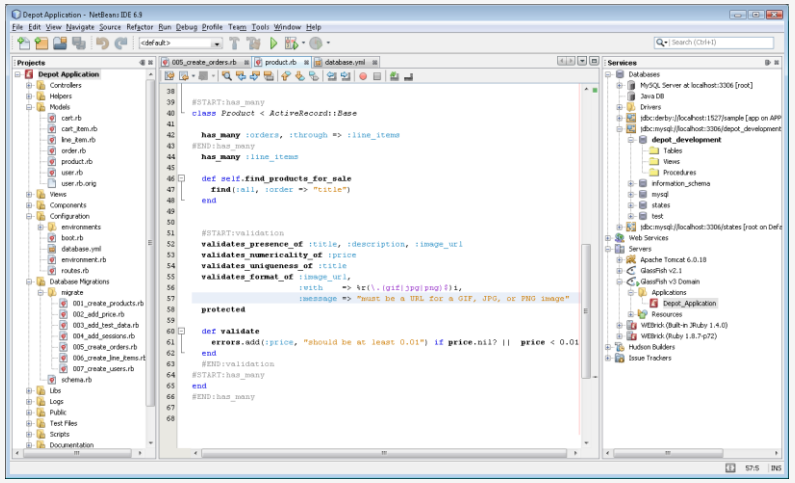
1. Język obiektowy
2. Niezależność od platformy
3. Prostota
4. Czy Java jest powolna?
5. Java jest "duża".
6. Podobieństwo do C#

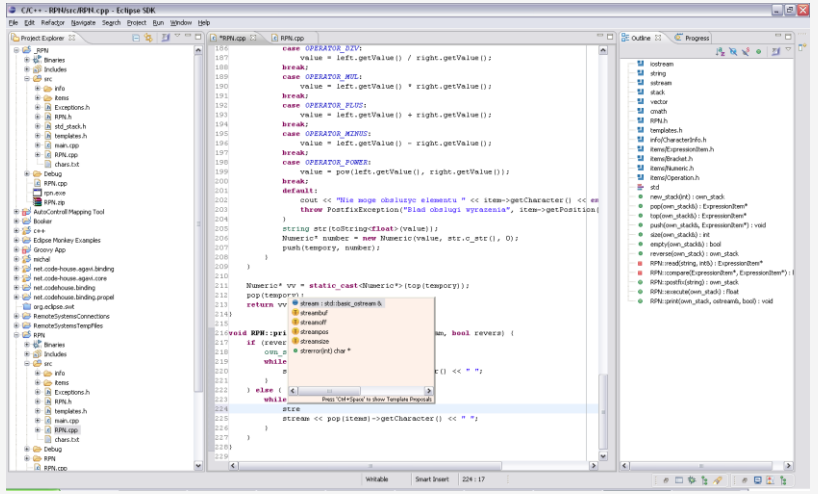
Środowisko do programowania

- Zainstalowane Java Development Kit w odpowiedniej wersji
  Sposób instalacji zależny od systemu operacyjnego. JDK można pobrać z strony Oracle lub OpenJDK.
  Ustawienie zmiennej środowiskowej JAVA_HOME

- Środowisko uruchomieniowe Java (JRE)
- Środowisko deweloperskie - IDE
  Darmowe narzędzia: IntelliJ IDEA, Eclipse, NetBeans







Pojęcia

Kod źródłowy programu - zapis programu komputerowego przy pomocy określonego języka programowania, opisujący operacje jakie powinien wykonać komputer na zgromadzonych lub otrzymanych danych. Kod źródłowy jest wynikiem pracy programisty i pozwala wyrazić w czytelnej dla człowieka formie strukturę oraz działanie programu komputerowego.

Kompilator - program, który kod źródłowy zamienia w kod

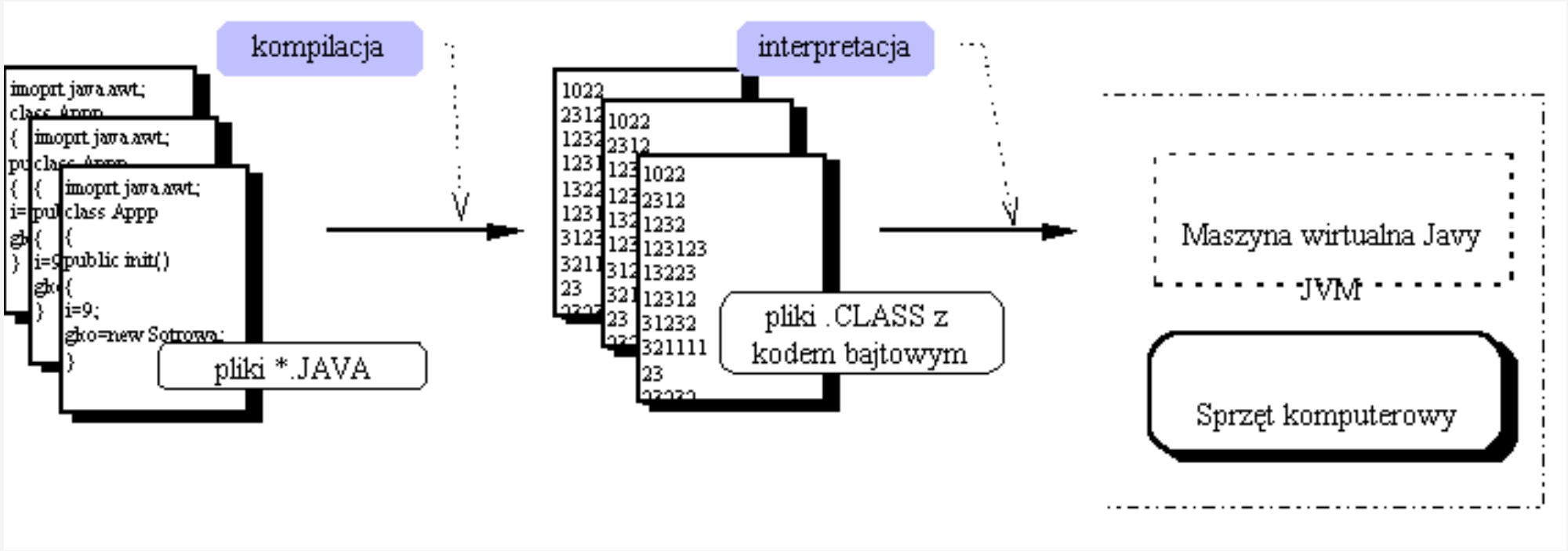
napisany w innym języku. Oprócz tego kompilator ma za zadanie

odnaleźć błędy leksykalne i semantyczne oraz dokonać optymalizacji kodu.

Kod bajtowy - bytecode, wynik kompilacji programu napisanego

w Javie, kod ten jest zrozumiały dla środowiska uruchomieniowego Java (JVM)

Jak działa język programowania



Zestaw znaków Unicode

Programy w języku Java pisze się przy wykorzystaniu znaków z zestawu Unicode. Znaków tych można używać w każdym miejscu programu, wliczając komentarze i identyfikatory, takie jak nazwy zmiennych.
W odróżnieniu od 7-bitowego zestawu ASCII, który zawiera tylko

znaki alfabetu języka angielskiego, i 8-bitowego zestawu ISO Latin-1, który zawiera tylko znaki najważniejszych europejskich języków, Unicode pozwala na wykorzystywanie praktycznie dowolnego języka będącego w użyciu na naszej planecie.

Rozpoznawanie wielkości znaków i białe znaki

W Javie wielkość liter ma znaczenie. Słowa kluczowe są pisane małymi literami i nie można odstępować od tej zasady. To znaczy, że napisy While i WHILE nie są tym samym co słowo kluczowe while.

Analogicznie, jeśli ktoś zdefiniuje w programie zmienną o nazwie i, to nie może się do niej odnosić za pomocą identyfikatora I.

Słowa zarezerwowane

Poniżej znajduje się lista słów zarezerwowanych w języku Java (należą one do składni języka i nie można nimi nazywać zmiennych, klas itd.):

```
abstract const    final      int       public       throw
assert   continue finally    interface return       throws
boolean  default  float      long      short        transient
break    do       for        native    static       true
byte     double   goto       new       strictfp     try
case     else     if         null      super        void
catch    enum     implements package   switch       volatile
char     extends  import     private   synchronized while
class    false    instanceof protected this
```

Identyfikatory

Identyfikator to nazwa nadana pewnej części programu w Javie, np. klasie, metodzie klasy albo zmiennej zadeklarowanej w metodzie. Identyfikator może mieć dowolną długość i składać się ze wszystkich znaków z zestawu Unicode, ale nie może zaczynać się od cyfry.
Zasadniczo identyfikatory nie mogą też zawierać znaków interpunkcyjnych. Wyjątkiem są znaki podkreślenia (_) i symbol dolara ($) oraz inne symbole walut Unicode.

Poniżej znajdują się przykłady poprawnych identyfikatorów w Javie:

i x1 aktualnaGodzina aktualna_godzina_i_data

Literały

Literały to wartości występujące bezpośrednio w kodzie źródłowym. Mogą być liczbami całkowitymi i zmiennoprzecinkowymi, pojedynczymi znakami w pojedynczych cudzysłowach, łańcuchami znaków w podwójnych cudzysłowach oraz zarezerwowanymi słowami true, false

i null.

Oto kilka przykładowych literałów:

1 1.0 '1' "jeden" true false null

Typy danych

W Javie podobnie jak w innych językach wyróżniamy wiele typów danych mogących przechowywać zarówno liczby stało i zmiennoprzecinkowe, znaki, ciągi znaków oraz typ logiczny. Java posiada ścisłą kontrolę typów, czyli mówiąc prościej każdy obiekt musi mieć określony typ.

byte - 1 bajt - zakres od -128 do 127

short - 2 bajty - zakres od -32 768 do 32 767

int - 4 bajty - zakres od -2 147 483 648 do 2 147 483 647

long - 8 bajtów - zakres od -2^63 do (2^63)-1 (posiadają przyrostek **L**, lub **l**)

float - 4 bajty - max ok 6-7 liczb po przecinku (posiadają przyrostek **F**, lub **f**)

double - 8 bajtów - max ok 15 cyfr po przecinku (posiadają przyrostek **D**, lub **d**)

char – odpowiada jednemu znakowi (np. literze), może przechowywać liczby całkowite z zakresu
od 0 do 65 535.

boolean – wartość *logiczna*, może przyjąć jedną z wartości **true** (oznaczającą prawdę) lub **false** (oznaczającą fałsz).

Typ boolean

Typ logiczny (boolean) reprezentuje tylko prawdę i fałsz, więc może mieć jedną z dwóch wartości reprezentujących dwa stany logiczne: włączenie-wyłączenie, tak-nie, prawda-fałsz.
W Javie stany te reprezentowane są przez zarezerwowane słowa true i false.

Typ char

Typ char reprezentuje jeden znak Unicode.

Literały znakowe w języku Java wpisuje się w pojedynczy cudzysłów (między apostrofami):

```
char c = 'A';
```

Typ char

Sekwencja specjalna	Reprezentowany znak
\b	Backspace
\t	Tabulator poziomy
\n	Nowy wiersz
\f	Wysuw strony
\r	Powrót karetki
\"	Podwójny cudzysłów
\'	Pojedynczy cudzysłów
\\	Ukośnik odwrotny
\ xxx	Znak z zestawu Latin-1 o jednostce kodowej xxx będącej ósemkową liczbą z przedziału od 000 do 377. Dopuszczalne są także formy \ x oraz \ xx, np. \ 0, ale nie zaleca się ich używania, ponieważ mogą sprawiać problemy w stałych łańcuchowych, gdy za sekwencją specjalną znajduje się zwykła cyfra. Formę tę z reguły odradza się na rzecz \uXXXX.
\u xxxx	Znak Unicode o jednostce kodowej xxxx będącej liczbą szesnastkową. Takie sekwencje specjalne mogą występować w każdym miejscu w programie, nie tylko w literałach znakowych i łańcuchowych.

Liczby całkowitoliczbowe

Typy całkowitoliczbowe to byte, short, int oraz long.

Literały każdego z tych typów tworzy się dokładnie tak, jak można się spodziewać, tzn. wpisując odpowiedni ciąg cyfr, przed którymi opcjonalnie można umieścić znak minus. Oto kilka

przykładów literałów całkowitoliczbowych:

0

1

123

-42000

Binarne literały całkowitoliczbowe poprzedza się znakami 0b i oczywiście mogą się one składać wyłącznie z cyfr 0 i 1.
W Javie można też wykorzystywać ósemkowe literały całkowitoliczbowe. Oznacza się je znakiem 0 z przodu i mogą one zawierać wszystkie cyfry oprócz 8 i 9. Używa się ich rzadko i powinno się ich unikać, jeśli to możliwe. Poniżej znajdują się przykłady prawidłowych literałów szesnastkowych, binarnych i ósemkowych:

```
0xff                    // dziesiętna wartość 255 wyrażona w formacie szesnastkowym
0377                    // ta sama liczba, tylko w formacie ósemkowym
0b0010_1111             // dziesiętna wartość 47 wyrażona w formacie binarnym
```

Literały całkowitoliczbowe są 32-bitowymi wartościami typu int, chyba że na końcu mają literę L lub l, w którym to przypadku są 64-bitowymi wartościami typu long:

```
1234      // wartość typu int
1234L     // wartość typu long
0xffL     // inna wartość typu long
```

Działania arytmetyczne na liczbach całkowitych w Javie nigdy nie powodują przepełnienia ani niedopełnienia w wyniku przekroczenia zakresu danego typu. Zamiast tego następuje zawinięcie

liczby.
Na przykład:

```
byte b1 = 127, b2 = 1;      // Największa wartość typu byte to 127.
byte sum = (byte)(b1 + b2);
// Suma tych wartości zawija się do –128, czyli najmniejszego bajta.
```

Typy zmiennoprzecinkowe

Liczby rzeczywiste w Javie są reprezentowane przez typy float i double.

Format literału zmiennoprzecinkowego w Javie ma następującą postać: opcjonalny ciąg cyfr, kropka i kolejny ciąg cyfr. Na przykład:

- 123.45
- 0.0
- .01

Literały zmiennoprzecinkowe można też wyrażać w notacji wykładniczej, czyli naukowej, która składa się z liczby, litery e lub E (wykładnik) i kolejnej liczby. Druga liczba reprezentuje potęgę liczby 10, przez którą należy pomnożyć pierwszą liczbę. Na przykład:

- 1.2345E02 *// 1,2345 * 10^2, czyli 123,45*
- 1e-6 *// 1 * 10^–6, czyli 0,000001*
- 6.02e23 *// Stała Avogadra: 6,02 * 10^23*

Literały zmiennoprzecinkowe domyślnie są wartościami typu double. Aby utworzyć literał zmiennoprzecinkowy typu float, należy na końcu dodać literę f lub F:

- double d = 6.02E23;
- float f = 6.02e23f;

Uwaga!

Literałów zmiennoprzecinkowych nie można wyrażać w notacji szesnastkowej, binarnej ani ósemkowej.

Konwertowanie typów prostych

W języku Java można wykonywać konwersje między typami całkowitoliczbowymi i zmiennoprzecinkowymi.

Ponadto dzięki temu, że każdemu znakowi odpowiada jakaś liczba z kodowania Unicode, wartości char również można konwertować na liczby całkowite i zmiennoprzecinkowe i odwrotnie. Tak naprawdę jedynym typem prostym, którego nie można przekonwertować na

inny typ prosty, jest boolean.

Wyróżnia się dwa podstawowe rodzaje konwersji.

- **Konwersja rozszerzająca** polega na zamianie

wartości jednego typu na szerszy typ, tzn. o większym zakresie dopuszczalnych wartości.

Na przykład Java automatycznie wykonuje konwersje rozszerzające, gdy programista przypisuje literał typu int do zmiennej typu double albo literał typu char do zmiennej typu int.

- **Konwersja zawężająca**. Polega ona na zamianie wartości jednego typu

na typ węższy. Tego rodzaju konwersje nie zawsze są bezpieczne. Na przykład w konwersji

wartości całkowitej 13 na typ byte nie ma nic złego, ale lepiej nie robić tego samego z liczbą

13000, ponieważ typ byte może przechowywać tylko liczby z przedziału od -128 do 127. Jako

że konwersja zawężająca może powodować utratę części danych, kompilator Javy blokuje takie wykryte przypadki nawet wtedy, gdy ten węższy zakres jest wystarczający do zapisania danej liczby:

```
int i = 13;
byte b = i; // Kompilator na to nie pozwoli.
```

Rzutowanie

Jeżeli chcesz wykonać konwersję zawężającą i masz pewność, że nie spowoduje to utraty danych ani precyzji, możesz do tego zmusić Javę za pomocą specjalnej konstrukcji o nazwie **rzutowanie**.

Aby wykonać rzutowanie, przed wartością do przekonwertowania należy umieścić w nawiasie nazwę typu, na który ma zostać ona przekonwertowana. Na przykład:

```
int i = 13;
byte b = (byte) i; // Wymusza konwersję wartości typu int na typ byte.
i = (int) 13.456; // Wymusza konwersję literału double na wartość typu int 13.
```

Rzutowania typów prostych najczęściej używa się do konwertowania wartości zmiennoprzecinkowych na liczby całkowite. Efektem takiego rzutowania jest po prostu opuszczenie części ułamkowej (tzn. wartość zmiennoprzecinkowa zostaje zaokrąglona w stronę zera, a nie do najbliższej liczby całkowitej). Do wykonywania innych rodzajów zaokrąglania służą metody Math.round(), Math.floor() i Math.ceil().

Zmienne

Deklaracja - określamy typ i nazwę zmiennej

Inicjalizacja - nadanie wartości zmiennej

```
public class Zmienne{
  public static void main(String[] args){

    int liczba; // Deklaracja

    liczba = 5; // Inicjalizacja
  }
}
```

Operatory matematyczne

- \+ dodaje 2 liczby
- \- odejmuje dwie liczby
- * znak mnożenia
- / dzielenie całkowite
- % reszta z dzielenia

```
int a = 17;
int b = 4;
int c = a+b; //=21
c = a-b;     //=13
c = a*b;     //=68
c = a/b;     //=4 ponieważ 4*4=16 i zostaje reszty 1
c = a%b;     //=1 reszta z dzielenia
```

Operatory porównawcze

== sprawdza równość

!= różny

>= większy równy

<= mniejszy równy

> , < większy, mniejszy

Nazwa	Język Java	Opis
i	&&	Iloczyn logiczny - wszystkie wartości muszą być prawdziwe, aby została zwrócona prawda.
lub	||	Suma logiczna - co najmniej jedna z wartości musi być prawdziwa, aby została zwrócona prawda.
negacja	!	Zanegowanie wartości - czyli zwrócenie wartości przeciwnej.

Kolejność wykonywania działań

P	Ł	Operator	Typy argumentów	Działanie
16	L	`.`	obiekt, składowa	Dostęp do składowej obiektu
		`[ ]`	tablica, `int`	Dostęp do elementu tablicy
		`( argumenty )`	metoda, lista argumentów	Wywołanie metody
		`++, --`	zmienna	Postinkrementacja, postdekrementacja
15	P	`++, --`	zmienna	Preinkrementacja, predekrementacja
		`+, -`	liczba	Jednoargumentowy plus, jednoargumentowy minus
		`~`	liczba całkowita	Dopełnienie bitowe
		`!`	`boolean`	Logiczna negacja
14	P	`new`	klasa, lista argumentów	Utworzenie obiektu
		`( typ )`	typ, dowolny	Rzutowanie (konwersja typu)
13	L	`\*, /, %`	liczba, liczba	Mnożenie, dzielenie, reszta z dzielenia
12	L	`+, -`	liczba, liczba	Dodawanie, odejmowanie
		`+`	łańcuch, dowolny	Łączenie łańcuchów
11	L	`<<`	liczba całkowita, liczba całkowita	Przesunięcie w lewo
		`>>`	liczba całkowita, liczba całkowita	Przesunięcie w prawo z rozszerzeniem znaku
		`>>>`	liczba całkowita, liczba całkowita	Przesunięcie w prawo z rozszerzeniem zer
10	L	`<, <=`	liczba, liczba	Mniejszość, mniejszy lub równy
		`>, >=`	liczba, liczba	Większość, większy lub równy
		`instanceof`	typ referencyjny	Porównywanie typów
9	L	`==`	typ prosty, typ prosty	Równość (czy wartości są identyczne)
		`!=`	typ prosty, typ prosty	Nierówność (czy wartości są różne)
		`==`	typ referencyjny, typ referencyjny	Równość (odnoszą się do tego samego obiektu)
		`!=`	typ referencyjny, typ referencyjny	Nierówność (odnoszą się do różnych obiektów)
8	L	`&`	liczba całkowita, liczba całkowita	Iloczyn bitowy
		`&`	`boolean, boolean`	Iloczyn logiczny
7	L	`^`	liczba całkowita, liczba całkowita	Bitowa alternatywa wykluczająca (XOR)
		`^`	`boolean, boolean`	Logiczna alternatywa wykluczająca (XOR)
6	L	`|`	liczba całkowita, liczba całkowita	Suma bitowa
		`|`	`boolean, boolean`	Suma logiczna
5	L	`&&`	`boolean, boolean`	Warunkowy iloczyn logiczny
4	L	`||`	`boolean, boolean`	Warunkowa suma logiczna
3	P	`? :`	`boolean`, dowolny	Operator warunkowy (trójargumentowy)
2	P	`=`	zmienna, dowolny	Przypisanie
		`*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=`	zmienna, dowolny	Przypisanie z działaniem
1	P	`->`	lista argumentów, treść metody	Wyrażenie lambda

Łączność

Łączność to cecha operatora określająca sposób wykonania wyrażeń, które mogłyby być niejednoznaczne.

Ma największe znaczenie w wyrażeniach zawierających kilka operatorów o takim samym priorytecie.

Większość operatorów ma łączność lewostronną, co znaczy, że działania są wykonywane po kolei od lewej. Ale operatory przypisania i jednoargumentowe mają łączność prawostronną.

W kolumnie *Ł* na poprzednim slajdzie podano łączność każdej grupy operatorów. Wartość *L* oznacza łączność lewostronną, a *P* — prawostronną.

Operatory addytywne mają łączność lewostronną, więc wyrażenie a+b-c zostanie obliczone od lewej jako (a+b)-c. Operatory jednoargumentowe i przypisania są wykonywane od prawej.

Spójrz na poniższe wyrażenie złożone:

a = b += c = -~d

Zostanie ono obliczone następująco:

a = (b += (c = -(~d)))

Zadanie

Napisać program służący do konwersji wartości temperatury podanej w stopniach Celsjusza na stopnie w skali Fahrenheita
(stopnie Fahrenheita = 1.8 * stopnie Celsjusza + 32.0)

Zadanie

Wczytać od użytkownika 3 liczby całkowite i wypisać na ekran największą oraz najmniejszą z nich.

Instrukcja if-else

Instrukcja if jest podstawową instrukcją sterującą, za pomocą której program może podejmować decyzje, a dokładniej mówiąc, wykonywać różne instrukcje w zależności od pewnych warunków.

Z instrukcją if powinny być związane wyrażenie i instrukcja. Jeżeli wartością wyrażenia jest true, interpreter wykonuje instrukcję. Jeśli wartością wyrażenia jest false, interpreter pomija instrukcję.

```
if(age > 18){
    System.out.println("You are adult");
}
```

Instrukcja if może opcjonalnie zawierać słowo kluczowe else, po którym wpisuje się drugą instrukcję. W takim przypadku najpierw sprawdzana jest wartość wyrażenia i jeśli jest to true,zostaje wykonana pierwsza instrukcja, a jeżeli false — druga. Na przykład:

```
if (age > 18)
    System.out.println("You are adult");
else {
    System.out.println("You are not adult");
}
```

Klauzula else if

Instrukcja if-else służy do sprawdzania warunku i wybierania na podstawie wyniku tego testu jednej z dwóch instrukcji lub jednego z dwóch bloków kodu. Ale co zrobić, gdy trzeba wybrać

jeden z większej liczby bloków kodu? W takim przypadku najczęściej używa się klauzuli else if, która tak naprawdę nie jest żadną nową konstrukcją składniową, a jedynie często spotykanym

sposobem wykorzystania standardowej instrukcji if-else. Wygląda to tak:

```
if (n == 1) {
    // wykonuje pierwszy blok kodu
} else if (n == 2) {
    // wykonuje drugi blok kodu
} else if (n == 3) {
    // wykonuje trzeci blok kodu
}else{

}
```

Zadanie - BMI

Napisać program, który oblicza wartość współczynnika BMI (ang. body mass index) wg. wzoru: waga wzrost2 . Jeżeli wynik jest w przedziale (18,5 - 24,9) to wypisuje ”waga prawidłowa”, jeżeli poniżej to ”niedowaga”, jeżeli powyżej ”nadwaga”.

Instrukcja switch

Instrukcja if powoduje rozgałęzienie w wykonywaniu programu. Do tworzenia wielodrożnych rozgałęzień, jak powyżej, można używać wielu takich instrukcji. Ale nie zawsze takie rozwiązanie jest najlepsze, zwłaszcza gdy wszystkie gałęzie zależą od wartości jednej zmiennej.

W takim przypadku powtarzanie testu tej samej zmiennej w wielu instrukcjach if jest nieefektywne.

Lepszym rozwiązaniem jest zastosowanie instrukcji switch odziedziczonej z języka C.

```
switch (n) {
    case 1: // Rozpoczyna wykonywanie od tego miejsca, jeśli n == 1.
        // Wykonuje pierwszy blok kodu 1.
        break; // stop
    case 2: // Rozpoczyna wykonywanie od tego miejsca, jeśli n == 2.
        // Wykonuje drugi blok kodu.
        break; // stop
    case 3: // Rozpoczyna wykonywanie od tego miejsca, jeśli n == 3.
        // Wykonuje trzeci blok kodu.
        break; // stop
    default: // Jeśli żaden z pozostałych przypadków nie pasuje...
        // Wykonuje czwarty blok kodu.
        break; // stop
}
```

Zadania

Napisać program realizujący funkcje prostego kalkulatora, pozwalającego na wykonywanie operacji dodawania, odejmowania, mnożenia i dzielenia na dwóch liczbach rzeczywistych. Program ma identyfikować sytuację wprowadzenia błędnego symbolu działania oraz próbę dzielenia przez zero. Zastosować instrukcję switch do wykonania odpowiedniego działania w zależności od wprowadzonego symbolu operacji.

Scenariusz działania programu:

a) Program wyświetla informację o swoim przeznaczeniu.

b) Wczytuje pierwszą liczbę.

c) Wczytuje symbol operacji arytmetycznej: +, -, *, /.

d) Wczytuje drugą liczbę. e) Wyświetla wynik lub - w razie konieczności - informację o niemożności wykonania działania. f) Program kończy swoje działanie po naciśnięciu przez użytkownika klawisza Enter.

Instrukcja while

Instrukcja while to podstawowa konstrukcja języka Java do wykonywania powtarzalnych czynności.

Inaczej mówiąc, jest to jedna z podstawowych **konstrukcji pętlowych** tego języka. Jej składnia wygląda następująco:

while (*wyrażenie*)
Instrukcja

Poniżej znajduje się przykładowa pętla while drukująca liczby od 0 do 9:

```
int licznik = 0;
while (licznik < 10) {
        System.out.println(licznik);
        licznik++;
}
```

Instrukcja do

Pętla do jest bardzo podobna do instrukcji while, tylko jej wyrażenie pętlowe jest sprawdzane

na końcu, a nie na początku. To gwarantuje przynajmniej jednokrotne wykonanie treści pętli.

Jej składnia jest następująca:

do{

instrukcja

} while (*wyrażenie*);

Instrukcja for

Instrukcja for to konstrukcja pętlowa, która w niektórych przypadkach jest wygodniejsza w użyciu od pętli while i do.

Zawiera licznik,

czyli zmienną stanową, który jest inicjowany na początku, a następnie sprawdzany w celu dowiedzenia się, czy należy wykonać treść pętli, i zwiększany lub zmieniany w inny sposób po wykonaniu treści pętli i przed ponownym sprawdzeniem wyrażenia warunkowego. Inicjacja, test i modyfikacja to trzy podstawowe działania wykonywane na zmiennej pętlowej i w pętli for

for(*inicjacja*; *test*; *modyfikacja*) {

instrukcja

}

Poniższa pętla for drukuje liczby od 0 do 9, podobnie jak wcześniejsze pętle while i do:

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

Lub

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

Pętla for

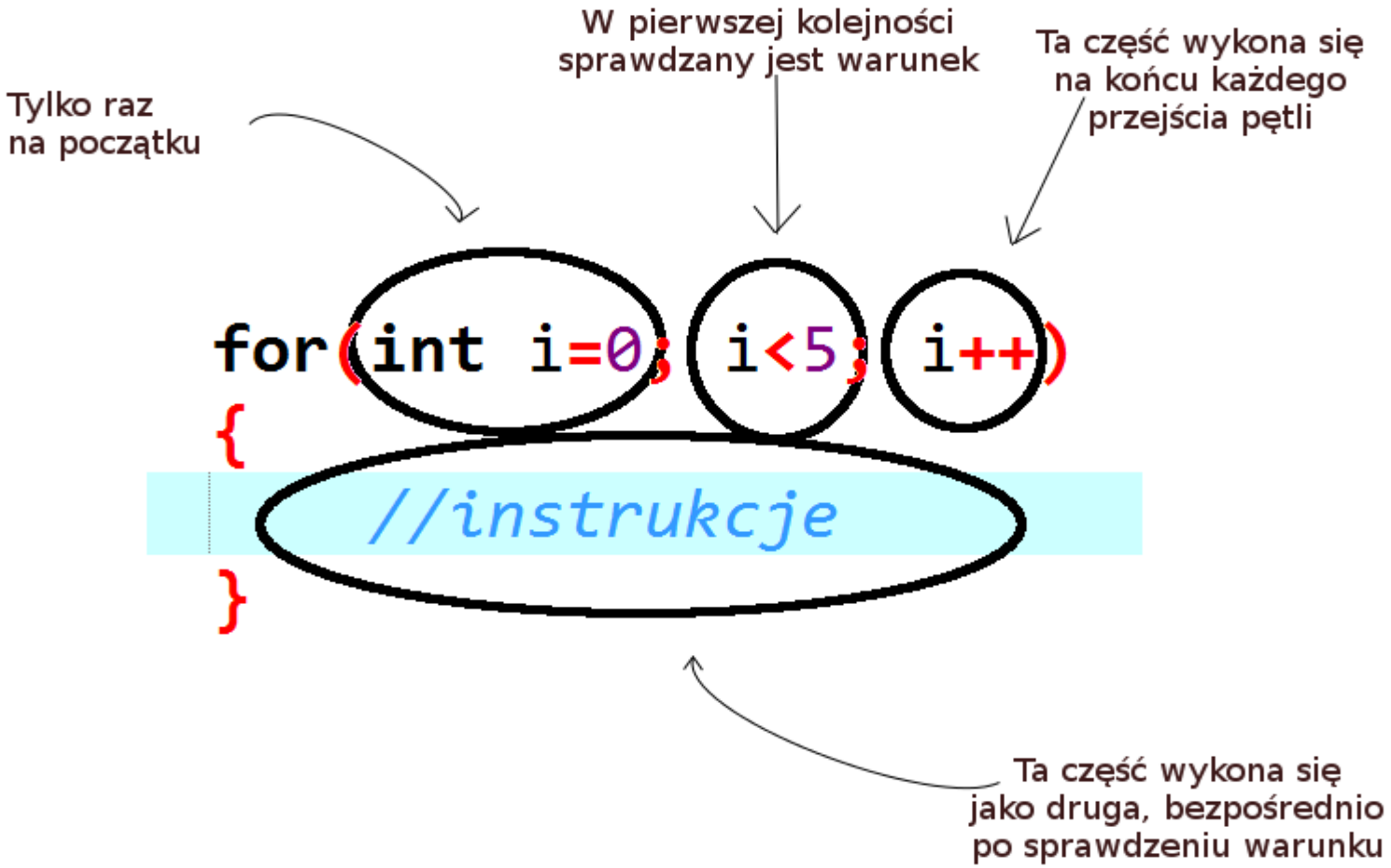
Schemat pętli for:

```
for(wyrażenie początkowe ; warunek ; modyfikator_licznika){
  instrukcje do wykonania
}
```

Przykład:

```
for(int i=0; i<10; i++){
    System.out.println("To jest pętla");
}
System.out.println("Koniec pętli");
```

Pętla for


Tylko raz
na początku
W pierwszej kolejności
sprawdzany jest warunek
Ta część wykona się
na końcu każdego
przejścia pętli
for(int i=0; i<5; i++)
{
//instrukcje
}
Ta część wykona się
jako druga, bezpośrednio
po sprawdzeniu warunku

Pętla while

```
while(warunek){
  instrukcje do wykonania
}
```

Pętlę while najczęściej wykorzystuje się w miejscach, gdzie zakładana ilość powtórzeń jest bliżej nieokreślona, ale znamy warunek jaki musi być spełniony. Jej schematyczną postać przedstawiono poniżej:

```
int licznik = 0;

while(licznik<10){
     System.out.println("To jest petla");
     licznik++;
}
System.out.println("Koniec pętli");
```

Pętla do while

Różni się ona od pętli while przede wszystkim tym, że to co znajduje się w jej wnętrzu wykona się przynajmniej raz, ponieważ warunek jest sprawdzany dopiero w drugiej kolejności.

```
do{
  instrukcje do wykonania
}
while(warunek);
```

Przykład:

```
int licznik = 0;

do{
    System.out.println("To jest petla");
    licznik++;
}
while(licznik<10);
System.out.println("Koniec pętli");
```

Przydatne rzeczy

Pobranie liczby od użytkownika:

```
int liczba;
Scanner wejscie = new Scanner(System.in);
liczba = wej.nextInt();
```

Pierwiastek z liczby:

```
int wynik = Math.sqrt(9); // oblicza pierwiastek z
liczby 9
```

Tablice

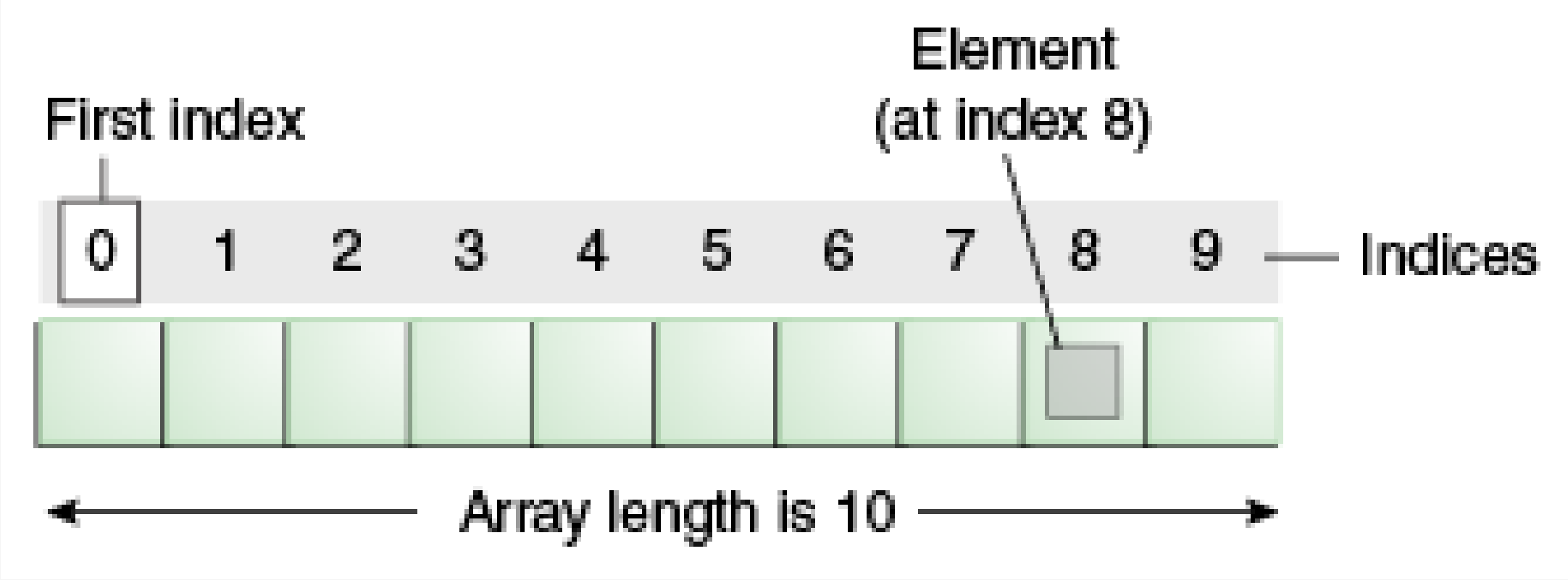
Tablica (ang. *array*) to specjalny rodzaj obiektu, w którym można przechowywać zero lub więcej wartości

Wartości te są przechowywane w **elementach**

tablicy będących nienazwanymi zmiennymi, do których można się odwoływać przy użyciu tzw. *indeksów* określających ich pozycję. O typie tablicy decyduje *typ zapisanych w niej elementów*

i wszystkie elementy tablicy muszą być jednego typu.

Tablica




```
int[] tablica = new int[10];
```

Tablice jednowymiarowe

Tablice to struktury, które pozwalają nam gromadzić większą ilość danych w uporządkowanej formie. Jeśli potrzebujemy przechować 100 imion, czy liczb zamiast deklarować 100 zmiennych możemy do tego użyć tablicy. W Javie istnieją zarówno tablice jedno jak i wielowymiarowe.

```
typ[] nazwa_tablicy = new typ[liczba_elementów];
```

lub

```
typ nazwa_tablicy[] = new typ[liczba_elementów];
```

Istnieje również inny sposób utworzenia tablicy, jeśli od razu znamy elementy:

```
typ[] tablica = {wartosc1, wartosc2, wartosc3, ...};
```

Typy tablic

Typy tablic to typy referencyjne, podobnie jak klasy. Egzemplarze tablic też są obiektami.

W odróżnieniu od klas typów tablicowych nie trzeba definiować. Wystarczy za nazwą typu

elementu wpisać nawias kwadratowy. Na przykład poniżej znajdują się trzy deklaracje zmiennych

typu tablicowego:

byte b; *// byte to typ prosty*

byte[] arrayOfBytes; *// byte[] to typ tablicowy: tablica bajtów*

byte[][] arrayOfArrayOfBytes; *// byte[][] to inny typ niż poprzedni: jest to tablica wartości typu byte[]*

String[] points; *// String[] jest tablicą obiektów typu String*

Tworzenie i inicjowanie tablic

Aby utworzyć wartość tablicową w języku Java, należy użyć słowa kluczowego new, podobnie jak w przypadku tworzenia obiektu. Typy tablicowe nie mają konstruktorów, ale przy tworzeniu ich wartości należy określić ich długość. Długość tablicy określa się za pomocą nieujemnej liczby całkowitej, którą należy wpisać w nawiasie kwadratowym:

```
// Tworzy nową tablicę do przechowywania 1024 bajtów.
byte[] buffer = new byte[1024];
// Tworzy tablicę 50 referencji do łańcuchów.
String[] lines = new String[50];
```

Inicjatory tablic

Aby utworzyć tablicę i zainicjować jej elementy w jednym wyrażeniu, należy opuścić długość tablicy i za nawiasem kwadratowym wpisać rozdzielaną przecinkami listę wyrażeń w klamrze.
Oczywiście typ każdego z tych wyrażeń musi dać się przypisać do typu elementu tablicy.

```
String[] greetings = new String[] { "Witaj", "Cześć", "Powitanko" };
int[] smallPrimes = new int[] { 2, 3, 5, 7, 11, 13, 17, 19, };
```

Sposoby dostępu do elementów tablicy

Elementy tablicy są zmiennymi. Jeżeli element tablicy znajduje się w wyrażeniu, to zostaje zastąpiony przechowywaną w nim wartością. A gdy element tablicy znajduje się po lewej stronie operatora przypisania, zostaje w nim zapisana nowa wartość. Ale w odróżnieniu od zwykłych

zmiennych elementy tablicy nie mają nazw, tylko numery. Dostęp do elementów tablicy można uzyskać za pomocą nawiasu kwadratowego.

```
// utworzenie tablicy dwóch łańcuchów
String[] responses = new String[2];
responses[0] = "Tak"; // ustawia pierwszy element tablicy
responses[1] = "Nie"; // ustawia drugi element tablicy
```

Granice tablicy

Należy pamiętać, że pierwszy element tablicy to a[0], drugi to a[1], a ostatni to a[a.length-1].

Często popełnianym błędem przez użytkowników tablic jest zastosowanie za małego (ujemnego) lub za dużego (większego od długości tablicy lub jej równego) indeksu.

Jeżeli indeks tablicy jest za mały lub za duży, zostaje zgłoszony

wyjątek ArrayIndexOutOfBoundsException.

Iterowanie tablic

Programiści często piszą pętle iterujące po kolei przez elementy tablicy w celu wykonania na nich pewnych działań. Najczęściej do tego celu używa się pętli for. Poniższy przykładowy kod oblicza sumę zapisanych w tablicy liczb całkowitych:

```
int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
int sumOfPrimes = 0;
for(int i = 0; i < primes.length; i++)
sumOfPrimes += primes[i];
```

Powyższa pętla ma bardzo typową strukturę, którą można spotkać w wielu programach. Ponadto w Javie dostępna jest też składnia typu foreach, o której była już mowa wcześniej. Przy jej użyciu kod sumujący można by było zapisać zwięźlej w następujący sposób:

```
for(int p : primes) sumOfPrimes += p;
```

Tablice

Pamiętajmy, że do utworzenia tablicy potrzebny jest operator **new**

```
int[] tablica = new int[100];
```

Powyższa instrukcja tworzy i inicjuje tablicę, w której można zapisać 100 liczb całkowitych. Elementy tablicy są **numerowane od 0** (w przypadku wyżej od 0 do 99)

Przypisywanie wartości do tablicy

```
int[] tablica = new int[5]; //deklaracja tablicy

tablica[0] = 23; //przypisanie wartosci
tablica[1] = 232;
tablica[2] = 242;
tablica[3] = 1;
tablica[4] = 41;

System.out.println(”Wartosc tablicy w miejscu o indeksie 2 =”  + tablica[2]);
```

Wypełnienie tablicy wartościami od 0 do 99 – pętla for

```
int[] tablica = new int[100];


for(int i=0; i<100; i++)
{
    tablica[i] = i; //zapełnia tablice wartościami od 0
do 99
}
```

Pętla typu for each

W języku Java dostępny jest bardzo użyteczny rodzaj pętli umożliwiającej przeglądanie tablic bez stosowania indeksów.

```
for (zmienna: kolekcja)
{
    Instrukcje
}
```

For each przykład

```
int[] tablica = new int[2];
tablica[0] = 232;
tablica[1] = 23112;

for (int wartosc: tablica)
{
    System.out.println(„Wartosci: ” + wartosc);
}
```

Tablice wielowymiarowe

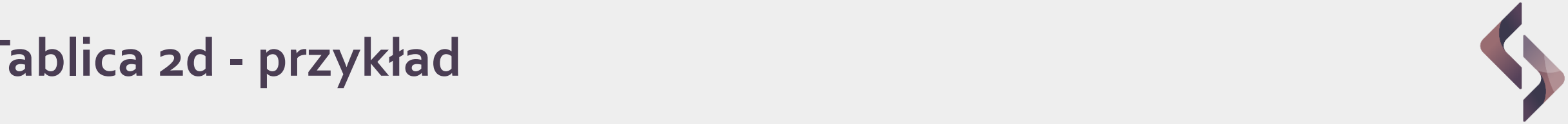
Różnią się przede wszystkim sposobem deklaracji i odwoływania do jej elementów. Za jej pomocą można na przykład w łatwy sposób wyobrazić sobie grę w statki - można przechowywać za pomocą współrzędnych miejsce położenia statków. Tablice wielowymiarowe w języku Java to tak naprawdę tablice tablic.

```
typ[][] nazwa_tablicy; //deklaracja
nazwa_tablicy = new typ[liczba1][liczba2]; //przypisanie (utworzenie)
typ[][] nazwa_tablicy2 = new typ[liczba1][liczba2]; //deklaracja i przypisanie (utworzenie)

int[][] tablica = new int[3][3]; // deklarujemy siatke 3x3
tablica[2][1] = 5;
int zmienna = tablica[2][1];
```

Pomocna jest interpretacja tablicy 2d jako wiesz/kolumna

Tablica 2d - przykład



```
int[][] tablica2D = new int[10][10];
```

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										

Tablice 2d - przykład

Liczba rzędów i kolumn nie musi być taka sama:

```
int [][] tablica= new int[6][10]
```

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										

Tablice 2d = dwie pętle for

```
// Tworzenie tablicy
int[][] tablica2D = new int[10][10];
// pętla zewnętrzna generuje indeksy rzędów
for (int i = 0; i < 10; i++) {
    // pętla wewnętrzna generuje indeksy kolumn
    for (int j = 0; j < 10; j++) {
        // możemy tak wyliczyć kolejną liczbę, ponieważ
        // każdy rząd odpowiada kolejnej dziesiątce
        tablica2D[i][j]= i*10 + j;
    }
}
```

Metody

Metoda to występująca pod pewną nazwą sekwencja instrukcji, którą można wywołać w kodzie

źródłowym. Przy wywoływaniu metodzie przekazuje się zero lub więcej wartości zwanych

argumentami. Metoda wykonuje pewne działania i może, choć nie musi, zwracać ich wynik.

Definiowanie metod

Sygnatura metody wygląda następująco:

modyfikatory *typ_zwracany* *nazwa* (*lista_parametrów*) [throws *wyjątki*]

Sygnatura metody, która określa:

- nazwę metody *nazwa* ;
- liczbę, kolejność, typy oraz nazwy używanych przez metodę parametrów;
- typ wartości zwracanej przez metodę;
- wyjątki kontrolowane, które metoda może zgłaszać (sygnatura może też zawierać listę wyjątków niekontrolowanych, ale nie jest to wymagane);
- różne modyfikatory metody dostarczające dodatkowych informacji o niej.

Sygnatura metody zawiera wszystko, co trzeba wiedzieć, aby móc ją wywołać. Jest **specyfikacją** metody.

Za sygnaturą (specyfikacją metody) znajduje się treść metody (jej implementacja), którą stanowi sekwencja instrukcji w klamrze.

Przykłady

```java
public class Program {
    // Wszystkie programy w Javie zawierają metodę o tej nazwie i sygnaturze, która stanowi ich punkt początkowy.
    public static void main(String[] args) {
        if (args.length > 0) System.out.println("Witaj, " + args[0]);
        else System.out.println("Witaj, świecie");
    }

    // Ta metoda przyjmuje dwa argumenty typu double i zwraca wartość typu double.
    static double distanceFromOrigin(double x, double y) {
        return Math.sqrt(x * x + y * y);
    }

    //Oblicza sumę argumentów i wyświetla wynik
    public static void Calculate(int a, int b) {
        int result = a + b;
        System.out.println("Result is: " + result);
    }


}
```

Listy argumentów o zmiennej długości

Metody można tak deklarować, aby przyjmowały zmienną liczbę argumentów wywołania.

Ich angielska potoczna nazwa to metody ***varargs***. Zaliczają się do nich metoda drukująca tekst

sformatowany System.out.printf(), związane z nią metody format() z klasy String

```
public static int max(int first, int... rest) {
/* na razie opuszczono treść główną */
}
```

Aby zamienić sygnaturę ze zmienną liczbą argumentów na „prawdziwą” sygnaturę, wystarczy zamienić człon ... na [].

Należy pamiętać, że lista parametrów może zawierać maksymalnie

jeden wielokropek, który musi się znajdować na końcu tej listy.

Dodamy trochę kodu źródłowego do metody max():

```
public static int max(int first, int... rest) {
        int max = first;
        for(int i : rest) { // dozwolone, ponieważ rest w rzeczywistości jest tablicą
                if (i > max)
                        max = i;
        }
        return max;
}
```

Metody

W języku możemy definiować własne funkcje (metody), metody mogą przyjmować argumenty.

```
void metoda1(){
  System.out.println("Ta metoda nic nie zwraca, ale wyświetla ten tekst");
}

int metoda2(){
  return 2;  //ta metoda zwraca liczbę 2 typu int
}

String metoda3(){
  return "Jakis napis";  //ta metoda zwraca String "Jakis napis"
}
```

Własne metody definiujemy poza funkcją main, wywołać możemy je wewnątrz funkcji main.

PROGRAMOWANIE OBIEKTOWE

Programowanie obiektowe

- Sposób programowania oparty o modelowaniu rzeczywistych bytów w elementy posiadające stan i zachowania
- W programie następuje interakcja pomiędzy obiektami
- Ścisłe powiązanie danych z procedurami
- Dane to wszystko to co opisuje modelowany byt za pomocą wartości określonego typu (liczbowe, znakowe)
- Zachowania to zbiór funkcji określających odpowiedzialność tego bytu – co potrafi wykonać.

Klasa

- Klasa w języku Java stanowi definicję obiektu
- Składa się pól – opis stanu
- Oraz funkcji – opis zachowania
- Instancja klasy to obiekt

Dziedziczenie

- Jedno z głównych założeń obiektowego programowania

- Obiekt A może dziedziczyć po obiekcie B co oznacza, że obiekt A posiada pełną funkcjonalność obiektu B oraz może ją rozszerzyć (dodać nowe funkcjonalności) lub nadpisać (edytować istniejące funkcjonalności)

- Dziedziczenie prowadzi do powstawania drzew obiektów

- W języku Java wszystkie klasy dziedziczą po klasie Object.

Hermetyzacja

- Stan obiektu powinien być ukryty. Obiekty nie są uprawnione do zmiany danych wewnątrz innych obiektów
- Do zmiany stanu obiektu powinny służyć tylko i wyłącznie jego zachowania

- W języku Java realizujemy tą zasadę poprzez odpowiednie nadanie modyfikatorów widoczności.

Polimorfizm

- Wywołanie funkcji obiektu spowoduje zachowanie odpowiednie dla pełnego typu obiektu wywoływanego. Jeśli dzieje się to w czasie działania programu, to nazywa się to *późnym wiązaniem* lub *wiązaniem dynamicznym*.

Przykład:
Obiekty Pies i Kot dziedziczą po obiekcie Zwierzak. Wywołanie funkcji dajGlos() na obiekcie Zwierzak, spowoduje wykonanie odpowiedniego zadania („hau” lub „miau”) zależnie od tego jakiego konkretnego typu był obiekt Zwierzak.

Polimorfizm

Polimorfizmem w językach obiektowych nazywa się również sytuację,
gdzie dwie funkcje ze zbioru zachowań danego obiektu mają taką samą nazwę,
ale do ich wywołania potrzebne są różne parametry wejściowe (typ, ilość lub kolejność)

Inaczej nazwany *statycznym wiązaniem* lub *przeciążaniem metod*.

Przykład:
Obiekt Pies posiada dwie metody dajGlos, jedna, zakładająca nagrodę dla
psa za wykonanie polecenia, druga nie przewidująca takiej nagrody.

dajGlos() {..}

dajGlos(nagroda) {..}

Paczki - package

- Programy Javowe nie stanowią monolitu.
- Poleca się, aby klasy dzielić na pakiety według ich odpowiedzialności

- Słówko kluczowe package określa względną lokalizację

klasy w drzewie projektu, gdzie każdy kolejny poziom

jest oddzielony kropką.

- Pakiety pozwalają na uporządkowanie kodu programu

oraz zapobiegają konfliktom nazw.

Import

- Aby klasa mogła w swojej definicji wykorzystać inne klasy, konieczne jest zaimportowanie tej klasy poprzez instrukcję import …;
- Import wskazuje na pakiet z jakiego pochodzi potrzebna klasa.
- Możemy importować pojedyncze klasy.
- Możemy importować całe pakiety (import nazwapakietu.*)

Klasa w języku Java – ciało klasy

- Ciało klasy zawarte jest w nawiasach klamrowych
- Język Java nie pozwala tworzyć pól

lub metod znajdujących się poza klasą

- Ciało klasy stanowią pola (dane) oraz metody (zachowania)
- Każdy element klasy może posiadać inny modyfikator widoczność

Programowanie obiektowe – konstruktor klasy

- Specyficzne metody wewnątrz klasy, które są odpowiedzialne za utworzenie instancji klasy w odpowiedni sposób (utworzenie stanu)
- Klasa może posiadać dowolną liczbę konstruktorów, ważne, aby były tworzone zgodnie z zasadą przeciążania metod
- Jeśli nie został jawnie utworzony konstruktor, zostanie utworzony bezparametrowy konstruktor
- Jeśli klasa dziedziczy po innej to w konstruktorze powinien zostać wywołany konstruktor klasy nadrzędnej

Klasa w języku Java – konstruktor

- Konstruktor może odwoływać się do pól oraz metod
- Aby odwołać się do elementu klasy należy

użyć słówka kluczowego this

- Aby odwołać się do elementu klasy nadrzędnej

należy użyć słówka kluczowego super

public, private, protected

- Klasy, pola oraz funkcje mogą mieć zdefiniowany zakres widoczności,

pozwala to programiście ukryć część implementacji klasy, która zgodnie

z zasadą hermetyzacji nie powinna wyjść poza daną klasę.

- **Private** – element jest widziany tylko wewnątrz klasy.
- **Default** – brak użycia modyfikatora widoczności powoduje, że element jest widoczny dla klas tego samego pakietu.
- **Protected** – element jest widziany w klasach z tego samego pakietu oraz w klasach, które po niej dziedziczą
- **Public** – element jest widoczny na zewnątrz

Programowanie obiektowe – pola klasy

- Pole klasy opisuje w sposób określony co do typu jedną z własności klasy
- Definicja pola klasy składa się z:

<modyfikator widoczności> <typ> <nazwa>;

Np. `private int groupSize;`

Co oznacza, że pole o nazwie groupSize typu całkowitego jest prywatne.

- Pole klasy może mieć wartość domyślną:

```
private int groupSize = 0;
```

Klasa w języku Java – extends …

- Opcjonalne słówko kluczowe oznaczające że dana klasa rozsze

inną klasę, która musi tu zostać określona za pomocą nazwy

- Każda klasa Javy niejawnie dziedziczy po klasie Object
- Klasa może dziedziczyć po maksymalnie jednej klasie
- Możliwe jest dziedziczenie wielopoziomowe,

tzn. A extends B extends C

- Dzięki dziedziczeniu klasa otrzymuje funkcjonalność

klasy rozszerzonej o widoczności public i protected

Przykład

```
class nazwa {
//deklaracje pól
typ pole1;
...
typ poleN;
//deklaracje metod
typ metoda1(lista-parametrów) {
//treść metody
}
...
typ metodaM(lista-parametrów) {
//treść metody
}
}
```

Klasa, która zawiera tylko trzy pola danych:

```
class Pudelko {
double szerokosc;
double wysokosc;
double glebokosc;
}
```

Klasa w języku Java – metody

- Opis zachowań klasy stanowi zbiór metod (funkcji)
- W Javie definicja funkcji wygląda następująco:

<modyfikator widoczności> <zwracany typ> <nazwa>(lista parametrów)
public int getGroupSize();

- Lista parametrów może być pusta, lub zawierać dowolną liczbę elementów oddzielonych przecinkami. Każdy parametr musi mieć określony typ i nazwę (np. String name)
- Zwracany typ oznacza jakiego typu obiekt zostanie przez metodę zwrócony. Jeśli funkcja ma nie zwracać wartości należy ustawić jej zwracany typ jako ***void***.

Klasa w języku Java – metody - przykład

```
public String getFullName(String name, String surName) {
        return name + ” ” + surName;
}
```

- Ciało metody, podobnie jak każdy blok kodu w Javie znajduje się w nawiasach klamrowych.
- Jeśli metoda zwraca wartość (typ inny niż void) to w ciele metody musi znaleźć się zapis *return <obiekt>;*
- Pisanie kodu po słowie return spowoduje błąd kompilacji lub jego niewykonanie.
- W metodach, które nie zwracają wartości (typ void) użycie słówka ***return;*** powoduje natychmiastowe opuszczenie funkcji

Klasa w języku Java – static

- Omawiany do tej pory stan klasy jest charakterystyczny dla danej instancji klasy (rozmiar grupy charakterystyczny dla grupy)
- Język Java pozwala na definiowanie pól w klasie, które będą współdzielone przez wszystkie instancje. Wartość takiego pola zmieniona w ramach jednego obiektu, spowoduje zmianę w pozostałych

- Pole statyczne nie jest powiązane z konkretną instancją klasy
- Pole statyczne tworzymy poprzez dodanie słówka kluczowego static w definicji pola: private static String trainingName;

Static - przykład

```
class Test{
   static void zwieksz(int liczba){
       liczba++;
   }
}

class Main{
   public static void main(String[] args) {
       int a = 5;
       Test.zwieksz(a);
       System.out.println(a);
   }
}
```

Przykłady należy przepisywać i sprawdzać ich działanie. Nie polecam kopiowania kodu.

Klasa w języku Java – static

- Podobnie jak stan klasy tak jego zachowania, także są powiązane z konkretną instancją klasy.
- Metody statyczne pozwalają na wykonanie funkcji, nie tworząc instancji klasy
- Metody statyczne nie mogą korzystać z niestatycznych zachowań i stanu klasy (nie są wywoływane w kontekście konkretnej instancji).
- Elementy statyczne kody mogą być wywoływane z niestatycznej funkcji
- Dostęp do statycznych elementów można uzyskać poprzez obiekt lub bezpośrednio po odwołaniu do nazwy klasy

Klasa w języku Java – final

- Słówko kluczowe final zależnie od użycia może mieć różny kontekst
- Dla klas – oznacza, że klasa ta nie może zostać rozszerzona
- Dla metod – oznacza, że dana metoda nie może zostać nadpisana w klasie dziedziczącej
- Dla pól – oznacza, że obiekt przypisany do tej referencji nie zmieni się w cyklu życia obiektu (co nie oznacza że stan obiektu na jaki wskazuje referencja nie może się zmienić !).
- Połączenie słówek final i static powoduje powstanie ***stałej.***

private static final String POLISH_PHONE_CODE=„0048”;

Jak nazywać klasy, pola, metody? Konwencja

- Pola, metody, klasy, zmienne nie mogą nazywać się jak słówka kluczowe języka (np. return, ale returnAmount tak);
- Nazwy klas piszemy zgodnie z konwencją ***camelCase*** zaczynając od dużej litery (inicjał każdego wyrazu z dużej litery, reszta mała, nie używamy polskich znaków). Nazwa klasy powinna mówić co dana klasa modeluje.
- Nazwy metod piszemy zgodnie z konwencją camelCase zaczynając od małej litery. Nazwa metody powinna jednoznacznie określać co dana metoda ma robić (nazwy powinny być możliwie zwięzłe – max 65535 ☺).
- Nazwy pól podobnie jak metody nazywamy zgodnie z camelCase zaczynając z małej litery. Należy określić jaką wartość dane pole reprezentuje. W nazwie pola nie należy powielać nazwy klasy. (w klasie Cat, pole dotyczące koloru, może nazywać się color, a nie catColor).
- Nazwy zmiennych wewnątrz metod nazywamy podobnie jak pola.
- Do nazywania stałych należy używać wersalików z wyrazami oddzielonymi podkreśleniami, np. PI_NUMBER

Jak utworzyć nową instancję klasy?

Aby utworzyć nową instancję musimy wywołać konstruktor

- Każda klasa, w której nie został jawnie utworzony konstruktor posiada konstruktor bezparametrowy
- Chcąc utworzyć nowy obiekt klasy Cat i przypisać go do zmiennej cat muszę napisać kod:

```
Cat cat = new Cat();
```

- Jeśli chcemy skorzystać z innego konstruktora (o ile został utworzony) należy napisać kod:

```
Cat cat = new Cat(color);
```

Gdzie color to obiekt mówiący jakiego koloru jest kot.

Posiadam instancję klasy, jak odwołać się do pól/metod?

```
Cat cat = new Cat();
```

- Aby wywołać dowolną metodą na zmiennej cat musimy napisać kod:

```
cat.sleep();
cat.eat();
```

- Aby odwołać się do stanu (pola) zmiennej cat (o ile pozwala na to modyfikator widoczności!) należy napisać kod:

```
int age = cat.length;
```

Rozwiązaniem problemu hermetyzacji jest utworzenie metody zwracającej wartość pola:

```
int age = cat.getAge();
```

Typy danych w Javie – rozszerzenie

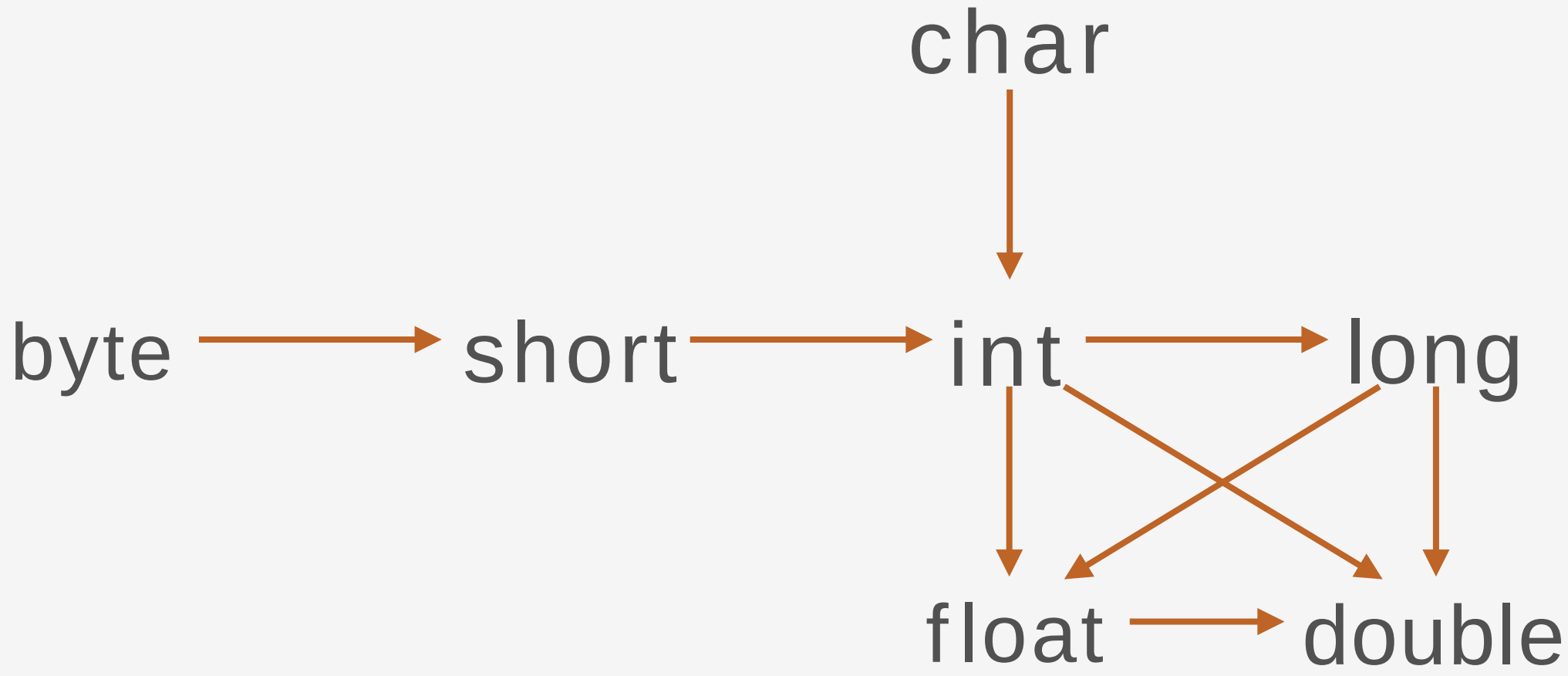
Typy podstawowe

- byte
- short
- int
- long
- float
- double
- boolean
- char
- void

Typy obiektowe

- Byte
- Short
- Integer
- Long
- Float
- Double
- Boolean
- Character
- Void

Typy danych - konwersje



Typ String

String jest jednym z najczęściej używanych typów zmiennej w Javie. Reprezentuje on łańcuch znakowy. String jest reprezentowany w kodzie przez łańcuch znakowy rozpoczęty i zakończony cudzysłowem.

Zmienną typu String możemy utworzyć na dwa sposoby:

- Przez konstruktor : String name = new String();
- Przez przypisane : String name = ”Adam”;

Klasa String posiada wiele wbudowanych funkcji (trim, replace, substring, length).

Aby utworzyć nową instancję klasy String poprzez sklejenie innych należy użyć operator +
Np. name + ” ” + surName;

Każda klasa posiada funkcję toString(), która umożliwia konwersję instancji do łańcucha znakowego (nie musi być jawnie określona).

Klasa Object

Każda klasa Java niejawnie dziedziczy po klasie Object.

Dzięki temu każda klasa niejawnie posiada zestaw metod opisanych w dokumentacji:

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Najistotniejsze w tym momencie są funkcje:

- hashCode() – wyliczająca (zawsze ten sam) wartość całkowitą stanowiącą skrót obiektu
- equals(Object object) – zwracającą zmienną typu boolean, określającą czy dane obiekty są sobie równe
- toString() – konwersja instancji do String'a
- finalize() – metoda wywoływana w momencie niszczenia instancji w pamięci przed odśmiecacz pamięci

Każdą z tych metod można nadpisać (a oprócz finalize – należy) w swojej klasie!

HashCode i Equals - kontrakt

W momencie porównania dwóch instancji klasy na początku liczone są wartości hashCode dla obu:

- Jeśli nie są takie same to oznacza że instancje z całą pewnością są różne
- Jeśli są takie same to należy wywołać metodę equals, aby dokładnie porównać instancje.

Kontrakt:

a.hashCode() == b.hashCode() nie oznacza że a.equals(b) == true

,ale

a.equals(b) ==> a.hashCode() == b.hashCode()

Przykład, a =4, b=35, funkcja hashCode liczy x mod 31

a.hashCode() równa się 4, b.hashCode() równa się 4, ale a nie równa się b

Interfejs

- Interfejs reprezentuje listę zachowań obiektu (tylko ich definicje, bez implementacji!)
- Interfejs może posiadać stałe
- Interfejs nie posiada stanu!
- Nie można utworzyć instancji interfejsu
- Interfejs nie posiada implementacji metod (odstępstwem są domyślne implementacje wprowadzone w Javie 8).
- Klasa może implementować interfejs – nie rozszerzać (nie używamy extends w stosunku do interfejsów).
- Interfejs może rozszerzać inny interfejs (wtedy używamy extends)
- Klasa, która implementuje interfejs musi posiadać implementacje wszystkich metod zawartych w interfejsie !

Interfejs

```java
package com.sdajava.interfejs;

import java.io.Serializable;

public interface MainInterface extends Serializable{

    void getGroupSize();

    default String getCity(){
        return "Honolulu";
    }
}
```

```java
package com.sdajava.interfejs;

public class MainInterfaceImpl implements MainInterface {
    @Override
    public void getGroupSize() {
        System.out.printf("10");
    }

    @Override
    public String getCity() {
        return "Torun";
    }
}
```

Definicja interfejsu jest podobna do definicji klasy, słowo kluczowe class zostaje zastąpione przez interface.

Pierwsza metoda ma określony typ, nazwę i listę parametrów, ale nie posiada ciała, które musi zostać zaimplementowane w klasie implementującej tej interfejs.

Druga metoda posiada domyślną implementację zwracającą pewną wartość. Może być nadpisana w klasie implementującej interfejs.
Interfejs rozszerza inny interfejs.

Klasa implementuje podany interfejs, co oznacza że musi posiadać nadpisane wszystkie funkcje interfejsu, które nie posiadają domyślnej implementacji.

Dodatkowo nadpisana została metoda getCity(), która w wypadku wywołania metody na instancji tego typu spowoduje zwrócenie wartości Torum zamiast Honolulu

Zadanie: Utwórz interfejs zawierający zbiór zachowań klasy z poprzedniego zadania. Utwórz chociaż jedną metodą z domyślną implementacją.

Klasy abstrakcyjne

- Klasa abstrakcyjna może posiadać stan
- Nie można utworzyć instancji klasy abstrakcyjnej
- Klasa abstrakcyjna może posiadać część metod zaimplementowanych, a część znanych tylko z sygnatury jak w przypadku interfejsu
- Klasa może rozszerzać klasę abstrakcyjną (poprzez extends), musi wtedy posiadać zaimplementowane wszystkie metody klasy abstrakcyjnej nie posiadające implementacji
- Klasa abstrakcyjna może rozszerzać inną klasę (niekoniecznie abstrakcyjną) lub implementować interfejs
- Dobrym zwyczajem jest rozpoczynanie nazwy klasy abstrakcyjnej od słówka Abstract

Klasy abstrakcyjne

```java
package com.sdajava.interfejs;

public abstract class AbstractMainClass
    implements MainInterface {

    private String courseName = "Java";

    public abstract String getTeacherName();

}
```

Definicja klasy abstrakcyjnej od definicji klasy różni się obecnością słówka kluczowego abstract.
Każda metoda, która ma zostać niezaimplementowana musi także w swojej sygnaturze posiadać słówko abstract oraz nie posiadać ciała.
Klasa abstrakcyjna może mieć stan.

```java
package com.sdajava.interfejs;

public class MainImpl
    extends AbstractMainClass{

    @Override
    public String getTeacherName(){
        return "Lukasz";
    }

    @Override
    public void getGroupSize(){

    }
}
```

Klasa rozszerzająca klasę abstrakcyjną posiada implementację abstrakcyjnej metody z klasy abstrakcyjnej oraz implmentację metody z interfejsu, który implementuje klasa abstrakcyjna!

Typy wyliczeniowe - enum

Typ zawierający listę wartości jaką zmienna danego typu może przyjąć,
np. status książki w wypożyczalni.

```
public enum BookStatus{
        BORROW,
        IN_LIBRARY,
        DESTROYED,
        LOST;
}
```

Do wartości enuma odwołujemy się jak do pola statycznego w klasie, tj.

```
BookStatus status = BookStatus.ZAGUBIONA;
```

Klasa anonimowa

Klasa anonimowa to klasa nie posiadająca nazwy. Definicja takiej klasy jest tworzona zawsze w momencie tworzenia nowej instancji. Klasy anonimowe najczęściej są jednokrotną, nigdzie indziej nie używaną implementacją klasy abstrakcyjnej lub interfejsu.

Klasa wewnętrzna

Klasa wewnętrzna to klasa, której definicja znajduje się wewnątrz innej klasy.
Klasa wewnętrzna może mieć stan, metody, konstruktory.

Jeśli klasa A znajduje się w klasie B, to odwołanie do niej następuje przez kropkę:
A.B.poleStatyczne

Utworzenie obiektu:
new A().new B();

Klasy wewnętrzne zależnie od modyfikatory widoczności mogą być nie widoczne na zewnątrz klasy otaczającej.

Adnotacje

Znacznik w kodzie, stanowi metainformację dla kompilatora, może być umieszczony na klasie, metodzie, polu, zmiennej.

Posiada formę @NazwaAdnotacji.

Popularne adnotacje:
@Override – oznacza, że metoda nadpisuje metodę z klasy nadrzędnej
@SuppressWarnings – oznacza, że programista jest świadomy zagrożeń w kodzie i prosi kompilator o wygaszenie ostrzeżeń.

Varargs

- Specyficzny sposób przekazywania parametrów do funkcji, kiedy znamy typ parametru, a chcemy pozostawić w dowolności liczbę tych parametrów, tzn. w jednym miejscu wywołamy funkcję z 2 parametrami, w innym z 5
- Definicja funkcji:

  public String concat(int numberPrefix, String… others)
- Operator … wskazuje że funkcja przyjmie dowolną wartość obiektów String
- Varargs zawsze jest ostatnim parametrem funkcji !
- W ciele funkcji varargs traktujemy jako tablicę typu określonego przed …
- Wywołanie funkcji:

  concat(1, ”Kot”, ”Pies”, ”Mysz”);

Najpopularniejszym przykładem varags jest parametr w funkcji main.

Typy generyczne

- Generyczność pozwala nam parametryzować klasy i metody
- Zwiększa uniwersalność klasy
- Jest to jeden ze sposobów na wprowadzenie polimorfizmu w kodzie
- Typ parametru metody lub typ pola klasy wybieramy dopiero w momencie jej użycia, a nie konstruowania
- Do użycia typów generycznych służy operator <>

Typy generyczne

- Generyczność możemy wprowadzać na poziomie klas i metod
- Klasa – generyczny kubek – nie wiemy co jest w środku, tworząc instancję musimy zadecydować czym będzie zawartość

Typy generyczne

- Klasa – generyczny kubek – w środku powinno znajdować się cokolwiek co jest napojem, ale zawsze napój tego samego typu

```
public class GenericCup <T extends Drink> {

    private T content;
}
```

Kubek może zostać sparametryzowany, aby przyjmować napój tylko konkretnego typu, np. mój ulubiony kubek na kawę.

W takim przypadku kubek może zawierać tylko kawę, nie możliwe jest wlanie do niego herbaty.

Aby utworzyć nowy kubek na kawę, musimy wywołać kod:

GenericCup<Coffee> coffeeCup = new GenericCup<>();

Klasa, która rozszerza generyczny kubek powinna określić jakiego typu zawartość będzie w kubku:

public class Mug extends GenericCup<Cofferr> { … }

Typy generyczne

- Metody – sparametryzować możemy zwracany typ:

public <T extends Drink> fillCup(String name, Class<T> type)
Konieczne jest przekazanie jako parametr funkcji typu generycznego (obiekt Class można pobrać z każdej klasy i obiektu poprzez wywołanie metody getClass())

- Parametry :

public static <T> void copy(List<T> dest, List<? extends T> src)
Składnie <? extends T> oznacza że akceptowany jest dowolny typ rozszerzający typ generyczny T.

Symbole typów generycznych oznacza się najczęściej dużą pojedynczą literą (T, R, S).
Klasa lub metoda może posiadać wiele różnych typów generycznych (wymienione po przecinku)

Typy generyczne - ograniczenia

- tworzyć obiektów typów sparametryzowanych (new T())
- używać operatora instanceOf (z powodu j.w.),
- używać ich w statycznych kontekstach (bo statyczny kontekst jest jeden dla wszystkich różnych instancji typu sparametryzowanego) – dotyczy klas
- wywoływać metod z konkretnych klas i interfejsów, które nie są zaznaczone jako górne ograniczenia parametru typu (w najprostszym przypadku tą górną granicą jest Object, wtedy możemy używać tylko metod klasy Object).

Wyjątki w Javie

- Wyjątki jest to mechanizm obsługi błędów w języku Java.
- Każdy wyjątek musi być obiektem klasy Throwable lub klasy z niej dziedziczącej.
- Wyjątki dzielimy na dwie grupy: checked i unchecked.
- Wyjątki checked to wyjątki dziedziczące po klasie Exception.
- Są to wyjątki, których wystąpienie musi zostać jawnie określone i muszą zostać obsłużone.
- Wyjątki unchecked to wyjątki dziedziczące po klasie RuntimeException.
- Są to wyjątki, których deklaracja i obsłużenie jest dobrowolnie.

Wyjątki w Javie – try catch

- Blok try – catch –finally zapewnia nam wykonanie bloku kodu w sposób zapewniający obsługę potencjalnych błędów
- try zawiera blok kodu, który może powodować błąd
- catch to blok zawierający kod, w którym obsługujemy błąd
- finally to blok kodu, który wykona się zawsze, nie zależnie czy w bloku try wystąpi błąd

Wyjątki w Javie – try catch

Bloków catch może być wiele, należy pamiętać aby je układać „od szczegółu do ogółu”.

Tj. jeśli zrobimy obsługę wyjątku nadrzędnego przed obsługą wyjątku dziedziczącego to wykona się blok kodu dla ogólniejszego wyjątku

Od Javy 7 bloki catch dla różnych wyjątków można łączyć za pomocą operatora |.

Bloki catch i finally są opcjonalne, ale przynajmniej jeden z nich musi się pojawić.

```
import java.util.Scanner;

public class Odczyt{
  public static void main(String[] args){
      int tab[] = {1,2,3,4,5};
      Scanner odczyt = new Scanner(System.in);
      int index = -1;

      System.out.println("Podaj indeks tablicy, który chcesz zobaczyć: ");
      index = odczyt.nextInt();

      try {
          System.out.println(tab[index]);
      } catch (ArrayIndexOutOfBoundsException e) {
          System.out.println("Niepoprawny parametr, rozmiar tablicy to: "+tab.length);
      }
  }
}
```

Wyjątki w Javie – throws a throw

- Throws jest elementem sygnatury funkcji, określa jakie wyjątki może rzucić ta funkcja. Standard języka wymaga zapisania wyjątków checked, zapisanie wyjątków unchecked jest dobrowolne (pisze się je tylko w uzasadnionej sytuacji, aby nie zaciemniać kodu).

```
public String showArrayElements(int[] i) throws IndexOutOfBoundsException
```

- Throw służy do wywołania wyjątku. Aby go wywołać należy utworzyć nową instancję.

```
throw new MyBusinessException();
```

Wywołanie wyjątku w naszym kodzie może spowodować konieczność dodania throws w sygnaturze metody.

Wyjątki nie muszą zostać obsłużone w metodzie, która wywołała błędną funkcję. Dodanie throws na tej metodzie przeniesie odpowiedzialność za obsługę błędów do metody wyżej.

Wyjątki w Javie – własne wyjątki

- Wyjątek jest zwykłą klasą
- Chcąc napisać własny wyjątek tworzymy klasę rozszerzającą Exception lub RuntimeException

- Własne wyjątki wykorzystujemy i obsługujemy jak te wbudowane w język Java

Wyjątki w Javie – co zawiera wyjątek

- **Z wyjątku możemy odczytać:**
  - Klasę wyjątku
  - Wiadomość zapisaną w wyjątku
  - Dodane pola (w przypadku naszego wyjątku polską wiadomość)
  - Stos wywołań funkcji (kolejne wywołania funkcji odłożone na stos – pomaga programiście w procesie debugowania kodu oraz określa miejsce gdzie nastąpiła sytuacja wyjątkowa)

  - Wyświetlenie stosu wywołań na standardowym wyjściu – funkcja printStackTrace();

Data i czas

- W języku Java mamy obecnie 3 implementacje daty: java.util.Date, java.sql.Data i java.time.LocalDate
- Ostatnia implementacja pochodzi z najnowszej wersji Javy i jest obowiązująca.
- LocalDate reprezentuje datę, LocalDateTime posiada również reprezentacje czasu
- LocalDateTime.now() – zwróci obiekt zawierający aktualną datę (komputera).
- new LocalDateTime() jak wyżej

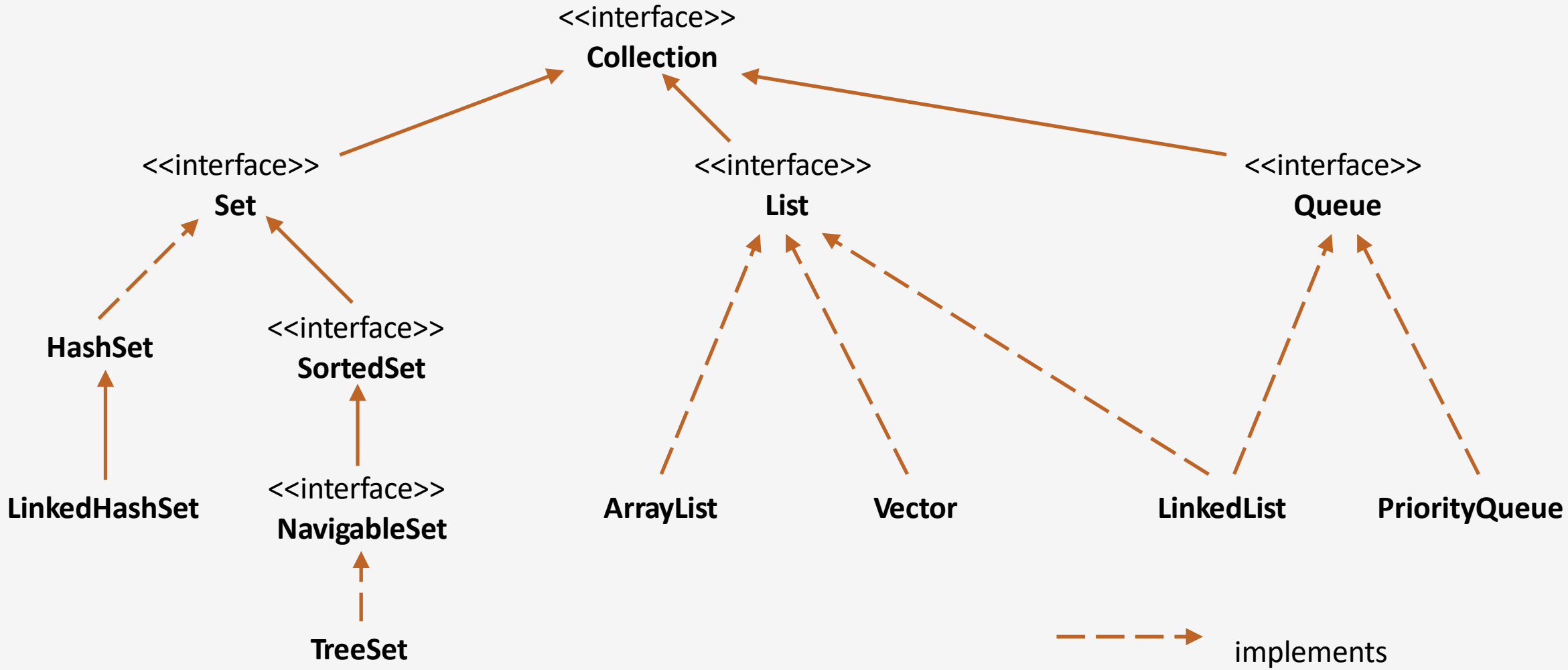
Data i czas - formater

- Obiekt daty może być tworzony z dowolnego napisu w dowolnym formacie
- Wystarczy przekazać obiekt formatera podczas parsowania daty.

```
DateTimeFormatter formatter =
        DateTimeFormatter.ofPattern(„dd-MM-yyyy");
LocalDate date = LocalDate.parse(input, formatter);

String formattedDate = date.format(formatter);
```

Kolekcje


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
HashSet
<<interface>>
SortedSet
LinkedHashSet
<<interface>>
NavigableSet
TreeSet
ArrayList
Vector
LinkedList
PriorityQueue
implements
extends

Kolekcje

- Wszystkie kolekcje w Javie implementują interfejs Collection
- Kolekcje służą jako magazyny obiektów
- Nie mogą przechowywać typów prostych (mogą obiekt po autoboxingu)
- W wyborze kolekcji należy uwzględnić do czego będzie wykorzystana, jakie operacje będą na niej najczęściej wykonywane
- Najbardziej znane kolekcje to List (ArrayList, LinkedList), Set (HashSet), Map (HashMap).

Kolekcje - List

- List – implementacje interfejsu List przechowują obiekty w postaci posortowanej listy, pozwalają przechowywać duplikaty

Klasa	Cechy	Zastosowanie
ArrayList	• w ‚tle' przechowuje dane w tablicy, której samodzielnie zmienia rozmiar wg potrzeb •dostęp do dowolnego elementu w czasie O(1) •dodanie elementu O(1), pesymistyczny przypadek O(n) •usunięcie elementu O(n)	Najczęstszy wybór z racji najbardziej uniwersalnego zastosowania. Inne imeplementacje mają przewagę tylko w bardzo specyficznych przypadkach. Jeśli nie wiesz, jakiej listy potrzebujesz, wybierz tą.
LinkedList	• przechowuje elementy jako lista powiązanych ze sobą obiektów (tj. pierwszy element wie, gdzie jest drugi, drugi wie, gdzie trzeci itd) •dostęp do dowolnego elementu O(n) (iteracyjnie O(1)) •dodanie elementu O(1) •usunięcie elementu O(1)	LinkedList ma przewagę w przypadku dodawania elementów pojedynczo, w dużej ilości, w sposób trudny do przewidzenia wcześniej, kiedy przejmujemy się ilością zajmowanej pamięci.W praktyce nie spotkałem się z sytuacją, w której LinkedList byłoby wydajniejsze od ArrayList
Vector	•Istnieje od początku Javy, z założenia miała to być obiektowa reprezentacja tablicy •Funkcjonalność i cechy analogiczne do ArrayList, ale znacznie mniej wyrafinowane	API oficjalnie zaleca korzystanie z klasy ArrayList zamiast Vector

Kolekcje - Set

- Set to kolekcja przechowująca unikalne wartości. Większość implementacji nie zachowuje kolejności.

Klasa	Cechy	Zastosowanie
HashSet	•zbiór nieposortowany •kolejność iteracji nieokreślona, może się zmieniać •dodanie elementu oraz sprawdzenie czy istnieje ma złożoność O(1) •pobranie kolejnego elementu ma złożoność O(n/h), gdzie h to parametr wewnętrzny (pesymistyczny przypadek: O(n))	Sytuacje, kiedy nie potrzebujemy dostępu do konkretnego elementu, ale potrzebujemy często sprawdzać, czy dany element już istnieje w kolekcji (czyli chcemy zbudować zbiór unikalnych wartości).W praktyce często znajduje zastosowanie do raportowania/zliczeń oraz jako ‚przechowalnia' obiektów do przetworzenia (jeśli ich kolejność nie ma znaczenia).
LinkedHashSet	•analogiczne, jak HashSet (dziedziczy po nim) •kolejność elementów używając iteratora jest deterministyczna i powtarzalna (zawsze będziemy przechodzić przez elementy w tej samej kolejności) •pobranie kolejnego elementu ma złożoność O(1)	Jesto to mniej znana i rzadziej stosowana implementacja, często może ona zastąpić HashSet poza specyficznymi przypadkami. W praktyce do iterowania w określonej kolejności często używane sa inne kolekcje (Listy, kolejki)
TreeSet	•Przechowuje elementy posortowane wg porządku naturalnego (jeśli implementują one interjfejs Comparable, w przeciwnym wypadku porządek jest nieokreslony) •Wszystkie operacje (dodanie, sprawdzenie czy istnieje oraz pobranie kolejnego elementu) mają złożoność O(log n)	Jeśli potrzebujemy, aby nasz zbiór był posortowany bez dodatkowych operacji (sortowanie nastepuje już w momencie dodania elementu) oraz iterować po posortowanej kolekcji.

Kolekcje - Map

- Map to kolekcja przechowująca pary klucz-wartość. Każdemu przechowywanemu w mapie obiektowi towarzyszy unikalny klucz (liczba, String, dowolny obiekt).

Klasa	Cechy	Zastosowanie
HashMap	•mapa nieposortowana •kolejność iteracji nieokreślona, może się zmieniać •dodanie elementu oraz sprawdzenie czy klucz istnieje ma złożoność O(1) •pobranie kolejnego elementu ma złożoność O(h/n), gdzie h to parametr wewnętrzny	Ogólny przypadek, tworzenie lokalnej pamięci podręcznej czy ‚słownika' o ograniczonym rozmiarze, zliczanie wg klucza.
LinkedHashMap	•analogiczne, jak HashMap (dziedziczy po niej) •kolejność kluczy używając iteratora jest deterministyczna i powtarzalna (zawsze będziemy przechodzić przez klucze w tej samej kolejności) •pobranie kolejnego elementu ma złożoność O(1)	Podobnie jak LinkedHashSet, rzadziej znana i stosowana. Przydatna, jeśli potrzebujemy iterować po kluczach w założonej kolejności.
TreeMap	•Przechowuje elementy posortowane wg porządku naturalnego kluczy (jeśli implementują one interjfejs Comparable, w przeciwnym wypadku porządek jest nieokreslony) •Wszystkie operacje (dodanie, sprawdzenie czy klucz istnieje oraz pobranie kolejnego elementu) mają złożoność O(log n)	Jeśli potrzebujemy, aby nasz zbiór był posortowany wg kluczy bez dodatkowych operacji (sortowanie nastepuje już w momencie dodania elementu) oraz iterować po posortowanej kolekcji.
Hashtable	• Historyczna klasa, która w Javie 1.2 została włączona do Java Collecion API	Oficjalnie zaleca się korzystanie z HashSet w większości wypadków zamiast Hashtable

Kolekcje – tworzenie i operacje

	List	Set	Map
Tworzenie	List<String> lista = new ArrayList<>();	Set<String> set = new HashSet<>();	Map<String, Object> mapa = new HashMap<>();
Dodawanie	lista.add(”obiekt1”); lista.addAll(innaLista);	set.add(”obiekt1”);	mapa.put(klucz, wartosc);
Usuwanie	lista.remove(obiekt); lista.removeAll(innaLista);	set.remove(obiekt);	mapa.remove(klucz);
Wyszukiwanie	lista.get(index)	Użycie klasy Iterator do pobrania kolejnego elementu. set.iterator().next();	mapa.get(klucz);

Kolekcje - iteracja

- Kolekcje podobnie jak tablice możemy w prosty sposób przeglądać za pomocą pętli foreach

- Dla listy i seta:

```
List<String> listaStringow = new ArrayList<>();
for (String element : listaStringow) {
        System.out.println(element);
}
```

- Dla mapy:

```
Map<String, String> map = new HashMap<>();
for (Map.Entry<String, String> entry : map.entrySet())
{
   System.out.println(entry.getKey() + "/" + entry.getValue());
}
```

WĄTKI

Proces to wykonujący się program wraz z dynamicznie przydzielanymi mu przez system zasobami (np. pamięcią operacyjną, zasobami plikowymi). Każdy proces ma własną przestrzeń adresową.

Systemy wielozadaniowe pozwalają na równoległe (teoretycznie) wykonywanie wielu procesów, z których każdy ma swój kontekst i swoje zasoby.

Wątek to sekwencja działań, która wykonuje się w kontekście danego procesu (programu)

Każdy proces ma co najmniej jeden wykonujący się wątek. W systemach wielowątkowych proces może wykonywać równolegle (teoretycznie) wiele wątków, które wykonują się jednej przestrzeni adresowej procesu.

WĄTKI

Równoległość działania wątków osiągana jest przez mechanizm przydzielania czasu procesora poszczególnym wykonującym się wątkom. Każdy wątek uzyskuje dostęp do procesora na krótki czas (kwant czasu), po czym „oddaje procesor” innemu wątkowi. Zmiana wątku wykonywanego przez procesor może dokonywać się na zasadzie:

współpracy (cooperative multitasking), wątek sam decyduje, kiedy oddać czas procesowa innym wątkom,

wywłaszczania (pre-emptive multitasking), o dostępie wątków do procesora decyduje systemowy zarządca wątków, który przydziela wątkowi kwant czasu procesora, po upływie którego odsuwa wątek od procesora i przydziela kolejny kwant czasu innemu wątkowi.

Java jest językiem wieloplatformowym, a różne systemy operacyjne stosują różne mechanizmy udostępniania wątkom procesora. Programy wielowątkowe powinny być tak pisane, by działały zarówno w środowisku „współpracy” jak i „wywłaszczania”

WĄTKI

Uruchamianiem wątków i zarządzaniem nimi zajmuje się klasa Thread.

Aby uruchomić wątek, należy utworzyć obiekt klasy Thread i dla tego obiektu wywołać metodę start().

Kod wykonujący się jako wątek – sekwencja działań wykonująca się równolegle z

innymi działaniami programu – określany jest przez obiekt implementujący interfejs

Runnable, który zawiera deklarację metody run(). Metoda run() określa to co ma robić wątek.

WĄTKI

Główne metody klasy Thread

1. Uruchamianie i zatrzymywanie wątków:
- start - uruchomienie wątku,
- stop – zakończenie wątku (metoda niezalecana),
- run - kod wykonywany w ramach wątku.

2. Identyfikacja wątków:
- currentThread - metoda zwraca identyfikator wątku bieżącego,
- setName - ustawienie nazwy wątku,
- getName -odczytanie nazwy wątku,
- isAlive - sprawdzenie czy wątek działa,
- toString - uzyskanie atrybutów wątku.

3. Priorytety i szeregowanie wątków:
- getPriority - odczytanie priorytetu wątku,
- setPriority - stawienie priorytetu wątku,
- yield - wywołanie szeregowania.

WĄTKI

4. Synchronizacja wątków:

- sleep zawieszenie wykonania wątku na dany okres czasu,
- join - czekanie na zakończenie innego wątku,
- wait - czekanie w monitorze,
- notify - odblokowanie wątku zablokowanego na monitorze,
- notifyAll - odblokowanie wszystkich wątków zablokowanych na monitorze,
- interrupt - odblokowanie zawieszonego wątku,
- suspend - zablokowanie wątku,
- resume - odblokowanie wątku zawieszonego przez suspend,
- setDaemon - ustanowienie wątku demonem,
- isDaemon - testowanie czy wątek jest demonem.

WĄTKI TWORZENIE

Możliwe są dwa sposoby tworzenia nowego wątku:

- poprzez dziedziczenie klasy Thread
- poprzez implementację interfejsu Runnable.

Sposób drugi stosujemy wówczas, gdy klasa reprezentująca wątek musi dziedziczyć

po innej niż Thread klasie (Java nie dopuszcza dziedziczenia wielobazowego).

Tworzenie wątku – dziedziczenie klasy Thread

Aby utworzyć wątek jako klasa potomna od klasy Thread należy:

1. Utworzyć nową klasę (na przykład o nazwie Tklasa) jako potomną klasy Thread i nadpisać metodę run() klasy macierzystej. Metoda ta ma zawierać kod do wykonania w ramach tworzonego wątku.

```
class TKlasa extends Thread {
....
 void run() {
 // Kod wątku
 }
....
}
```

2. Utworzyć obiekt nowej klasy Tklasa (na przykład thr):

```
TKlasa thr = new TKlasa(...);
```

3. Wykonać metodę start() klasy TKlasa dziedziczoną z klasy macierzystej:

thr.start();

Tworzenie wątku – implement. interfejsu Runnable

Aby utworzyć wątek korzystając z interfejsu Runnable należy:

1. Utworzyć nową klasę (np. o nazwie RKlasa) dziedziczącą po interesującej na innej klasie (np. o nazwie InnaKlasa) i implementującą interfejs Runnable. W ramach tej nowej klasy utworzyć metodę run(), która wykonywała będzie żądane czynności.

```
class RKlasa extends InnaKlasa implements Runnable {
      public void run() {
      // Zawartość metody run
      }
}
```

2. Utworzyć obiekt tej nowej klasy

```
Rklasa r1 = new RKlasa();
```

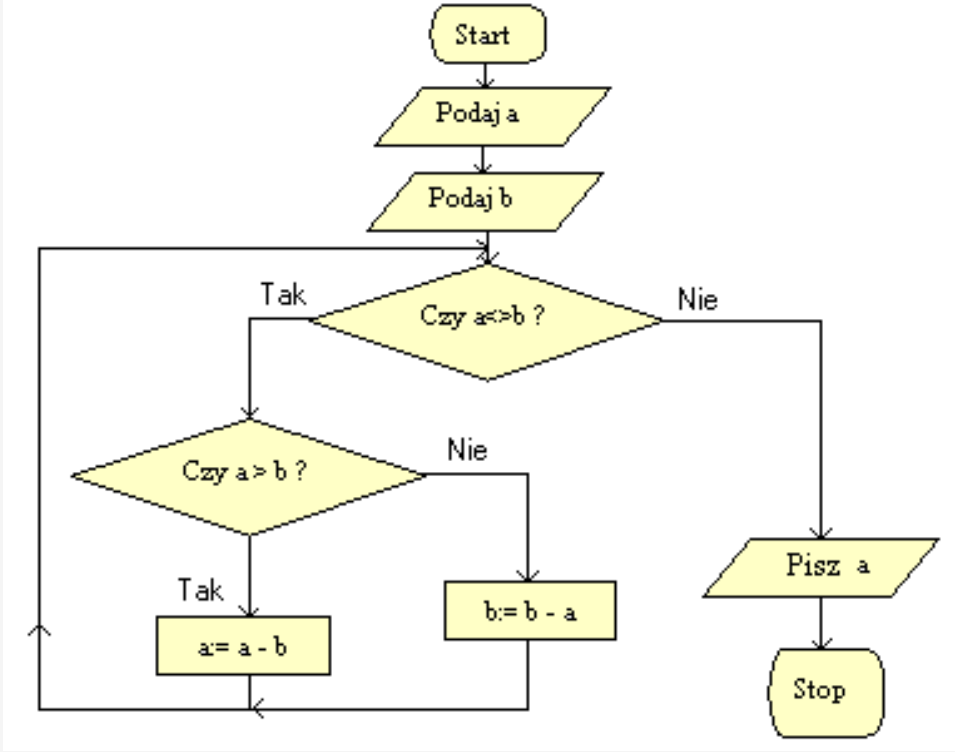
3. Utworzyć obiekt klasy Thread przekazując obiekt wcześniej utworzonej klasy jako parametr konstruktora klasy `Thread. Thread t1 = new Thread(r1);`

4. Uruchomić watek wykonując metodę start() klasy Thread. t1.start();

Algorytmy

Algorytm – skończony ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadań.

- Schematy blokowe
- Sztuczna Inteligencja
- Algorytmy genetyczne
- Sieci neuronowe



Klasyfikacja algorytmów

- dziel i zwyciężaj – dzielimy problem na kilka mniejszych, a te znowu dzielimy, aż ich rozwiązania staną się oczywiste;
- programowanie dynamiczne – problem dzielony jest na kilka, ważność każdego z nich jest oceniana i po pewnym wnioskowaniu wyniki analizy niektórych prostszych zagadnień wykorzystuje się do rozwiązania głównego problemu;
- metoda zachłanna – nie analizujemy podproblemów dokładnie, tylko wybieramy najbardziej obiecującą w danym momencie drogę rozwiązania;
- programowanie liniowe – oceniamy rozwiązanie problemu przez pewną funkcję jakości i szukamy jej minimum;
- poszukiwanie i wyliczanie – przeszukujemy zbiór danych, aż do odnalezienia rozwiązania;
- heurystyka – człowiek na podstawie swojego doświadczenia tworzy algorytm, który działa w najbardziej prawdopodobnych warunkach, rozwiązanie zawsze jest przybliżone.

Techniki implementacji

- proceduralność – algorytm dzielimy na szereg podstawowych procedur, wiele algorytmów współdzieli wspólne biblioteki standardowych procedur, z których są one wywoływane w razie potrzeby;
- praca sekwencyjna – wykonywanie poszczególnych procedur algorytmu, według kolejności ich wywołań, naraz pracuje tylko jedna procedura;
- praca wielowątkowa – procedury wykonywane są sekwencyjnie, lecz kolejność ich wykonania jest trudna do przewidzenia dla programisty;
- praca równoległa – wiele procedur wykonywanych jest w tym samym czasie, wymieniają się one danymi;
- rekurencja – procedura lub funkcja wywołuje sama siebie, aż do uzyskania wyniku lub błędu;
- obiektowość – procedury i dane łączymy w pewne klasy reprezentujące najważniejsze elementy algorytmu oraz stan wewnętrzny wykonującego je systemu;
- algorytm probabilistyczny – działa poprawnie z bardzo wysokim prawdopodobieństwem, ale wynik nie jest pewny,

Struktury danych

Struktura danych (ang. data structure) - sposób uporządkowania informacji w komputerze. Na strukturach danych operują algorytmy.

- rekord lub struktura (ang. record, struct), logiczny odpowiednik to krotka
- tablica
- lista
- stos
- kolejka
- drzewo i jego liczne odmiany (np. drzewo binarne)
- graf

Reprezentacja liczb

Najprostszym układem pozycyjnym jest system binarny. Elementami zbioru znaków systemu binarnego jest para cyfr: 0 i 1. Znak dwójkowy (0 lub 1) nazywany jest bitem. Liczby naturalne w systemie dwójkowym zapisujemy analogicznie jak w systemie dziesiętnym - jedynie zamiast kolejnych potęg liczby dziesięć, stosujemy kolejne potęgi liczby dwa.

$$9 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$9_{(10)} = 1001_{(2)}$$