

Język Java - podstawy programowania

Łukasz Kuszner
Krzysztof Giaro

Politechnika Gdańska, Wydział ETI
Katedra Algorytmów i Modelowania Systemów

9 stycznia 2010

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 1 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Java

Java jest obiektowym językiem programowania stworzonym przez grupę roboczą pod kierunkiem Jamesa Goslinga z firmy Sun Microsystems. Java jest językiem kompilowanym do tzw. bytecode'u, czyli postaci wykonywanej przez maszynę wirtualną.

Podstawowe koncepcje zostały przejęte z języka Smalltalk: maszyna wirtualna, automatyczne zarządzanie pamięcią (garbage collection) oraz z języka C++: duża część składni.

Uwaga: Javy nie należy mylić ze skryptowym językiem JavaScript, z którym ma niewiele wspólnego (głównie składnię podstawowych instrukcji).

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 2 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Powstanie języka Java

- 1990 - Bill Joy w raporcie "Further" sugeruje firmie SUN stworzenie środowiska obiektowego na bazie C++,
- 1991 - W ramach projektu "Green" powstaje język OAK - "Object Application Kernel" (James Gosling), przeznaczony dla aplikacji w elektronice powszechnego użytku,
- 1995 - zmiana nazwy na JAVA ze względu na zastrzeżenie nazwy OAK,
- 1996 - Pojawia się Netscape zgodny z Java 1.0, Sun propaguje darmowe środowisko JDK 1.0,
- 1999 - Java 2. (Java 1.2)
- 2000 - Dalszy dynamiczny rozwój języka, środowisk. Obecna stabilna wersja to 1.5, jest również wersja Beta 1.6.

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)

[◀](#) [▶](#)

[◀](#) [▶](#)

[Strona 3 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Środowisko Java

- Język obiektowy
- Kompilator do kodu pośredniego
- Maszyna Wirtualna Javy (JVM Java Virtual Machine)

Strona główna

Strona tytułowa

Spis treści



Strona 4 z 80

Powrót

Full Screen

Zamknij

Koniec

Motywacja twórców

Cele, które przyświecały twórcom języka można zawrzeć w kilku punktach:

- Java ma być nowoczesnym, łatwym do opanowania językiem obiektowym ogólnego przeznaczenia.
- Język i jego implementacja powinny zapewniać takie udogodnienia jak: silna kontrola typów, sprawdzanie zakresów tablic, wykrywanie prób użycia niezainicjalizowanych zmiennych i automatyczne zwalnianie pamięci (ang. garbage collection).
- Język ma zapewnić przenośność kodu jak i łatwość jego przyswojenia przez programistów znających C i C++.

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 5 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Zastosowania

- Tworzenie przenośnych aplikacji, w których wymagania czasowe nie są krytyczne
- Tworzenie aplikacji internetowych.
- Tworzenie aplikacji dla urządzeń mobilnych.

Strona główna

Strona tytułowa

Spis treści

◀◀ ▶▶

◀ ▶

Strona 6 z 80

Powrót

Full Screen

Zamknij

Koniec

Wydajność

Problemy z wydajnością zostały częściowo przełamane, gdy w JVM zamiast interpreterów kodu pośredniego języka Java zastosowano kompilację kodu pośredniego do kodu maszynowego w trakcie uruchomienia (ang. just-in-time compilation).

Strona główna

Strona tytułowa

Spis treści



Strona 7 z 80

Powrót

Full Screen

Zamknij

Koniec

Wydajność 2

Uwaga: Mimo że projektanci dołożyli starań aby aplikacje tworzone w Javie działały szybko i nie używały zbyt wielu zasobów pamięciowych, to należy pamiętać, że język nie został wymyślony po to, by konkurować pod tym względem z **C**, czy asemblerami.

Uwaga: W początkowym okresie nauki języka prawdopodobnie lepiej jest unikać narzędzi typu RAD (ang. Rapid Application Development), gdyż traci się wtedy często kontrolę nad zrozumieniem treści tworzonego programu.

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 8 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Java a C++

- **C++**: ufa programiście
- **Java**: chroni programistę (przed jego własnymi błędami)

Strona główna

Strona tytułowa

Spis treści



Strona 9 z 80

Powrót

Full Screen

Zamknij

Koniec

Java a C++ - cechy

- **C++:** Wiele cech i możliwości, język bardzo bogaty składniowo
- **Java:** Język łatwiejszy do nauczenia i opanowania

Strona główna

Strona tytułowa

Spis treści



Strona 10 z 80

Powrót

Full Screen

Zamknij

Koniec

Java a C++ - kompatybilność

- **C++:** Nadzbiór języka C (C jest kompatybilny z C++)
- **Java:** Kompatybilność z wcześniejszymi wersjami Javy

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



Strona 11 z 80

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Java a C++ - efektywność

- **C++:** Tworzenie efektywnego kodu wynikowego
- **Java:** Efektywne wykorzystanie nakładu pracy programisty

Strona główna

Strona tytułowa

Spis treści



Strona 12 z 80

Powrót

Full Screen

Zamknij

Koniec

Java a C++ - dostęp do pamięci

- **C++:** pełna kontrola dostępu do pamięci
- **Java:** dostęp do pamięci tylko za pomocą obiektów

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 13 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Java a C++ - różnice składniowe

- **C++:** W przeciwieństwie do Javy, możliwa redefinicja (przeciążanie operatorów), możliwe jest definiowanie szablonów, dziedziczenie po wielu klasach (ang. multiple inheritance)
- **Java:** W przeciwieństwie do C++, posiada wsparcie ze strony języka do tworzenia wielowątkowych aplikacji. Np. słowo kluczowe **synchronized** zapewnia wzajemne wykluczanie.

Strona główna

Strona tytułowa

Spis treści



Strona 14 z 80

Powrót

Full Screen

Zamknij

Koniec

Java a C++ - obiektowość

- **C++:** Język proceduralno-obiektowy, dopuszcza zmienne, stałe i funkcje poza klasami (na poziomie przestrzeni nazw)
- **Java:** język obiektowy (nie obiektami są tylko zmienne typów prostych), zmienne, stałe i funkcje mogą występować tylko w obrębie klas

Strona główna

Strona tytułowa

Spis treści

◀◀ ▶▶

◀ ▶

Strona 15 z 80

Powrót

Full Screen

Zamknij

Koniec

Java a C++ - przekazywanie parametrów

- **C++:** Możliwe jest wymuszenie przekazania parametrów przez wartość, bądź przez zmienną (wskaźnik)
- **Java:** Wszystkie parametry są przekazywane przez wartość, jednak obiekty są reprezentowane przez wskaźnik, a więc de facto przekazywane przez zmienną (warto pamiętać, że w Javie tablice są obiektami)

Strona główna

Strona tytułowa

Spis treści



Strona 16 z 80

Powrót

Full Screen

Zamknij

Koniec

Java a C++ - standardy

- **C++:** Istnieje standard języka
- **Java:** Nie został opracowany standard. Za "standard de facto" uważa się kolejne edycje środowiska firmy SUN.

Strona główna

Strona tytułowa

Spis treści



Strona 17 z 80

Powrót

Full Screen

Zamknij

Koniec

Maszyna Wirtualna

Maszyna wirtualna to ogólna nazwa dla programów tworzących środowisko uruchomieniowe dla innych programów. Maszyna wirtualna kontroluje wszystkie odwołania uruchamianego programu bezpośrednio do sprzętu lub systemu operacyjnego i zapewnia ich obsługę. Dzięki temu program uruchomiony na maszynie wirtualnej jest odizolowany od sprzętu na którym działa, pracuje na sprzęcie wirtualnym, "udawanym" przez odpowiednie oprogramowanie (maszynę wirtualną).

Wykonywanym programem może być zarówno pojedyncza aplikacja jak i cały system operacyjny lub nawet kolejna maszyna wirtualna. Są one zupełnie odizolowane przez maszynę wirtualną od maszyny fizycznej, w odróżnieniu od klasycznego systemu operacyjnego, który tylko zarządza uruchamianiem aplikacji na maszynie fizycznej.

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 18 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Implementacje Javy

- Implementacje bez dostępu do kodu źródłowego:
 - Sun Microsystems' Java HotSpot Virtual Machine for Windows, Linux and Solaris
 - Hewlett-Packard's Java for HP-UX, OpenVMS, Tru64 and Reliant(Tandem) UNIX UNIX platforms
 - IBM for MVS, AIX, OS/400, z/OS
 - Apple Computer MacOS Runtime for Java (MRJ)
- Implementacje typu Open source: AegisVM [1], Apache Harmony, CACAO, GCJ, IKVM.NET, Jamiga, JamVM, Jaos, JC, Jikes RVM, JNode, Jupiter JVM, Kaffe, leJOS, NanoVM, SableVM, JOP, Waba

Strona główna

Strona tytułowa

Spis treści



Strona 19 z 80

Powrót

Full Screen

Zamknij

Koniec

Narzędzia

- javac - kompilator,
- java - interpreter z konsolą,
- javaw - interpreter bez konsoli,
- javadoc - generator dokumentacji API,
- appletviewer - interpreter apletów,
- jar - zarządzanie plikami archiwów (JAR),
- jdb - debugger,
- NetBeans IDE - zintegrowane środowisko programistyczne (IDE) dla języka Java.

Strona główna

Strona tytułowa

Spis treści



Strona 20 z 80

Powrót

Full Screen

Zamknij

Koniec

Podstawy Javy na tle C++

- "Wszystko" co nie jest zmienną typu prostego jest obiektem, klasą lub składnikiem klasy - nie ma zmiennych lub funkcji globalnych.
- Klasy biblioteczne importujemy nie z plików nagłówkowych lecz pakietów (o hierarchicznych nazwach złożonych z oddzielonych kropkami członów). Deklaracja przynależności do pakietu powinna być pierwszą instrukcją w pliku, w przeciwnym razie działamy domyślnie w pakiecie "nienazwanym".
- Specyfikatory dostępu **public**, **protected**, **private** umieszcza się przed elementami klasy. Sama klasa może być dostępna publicznie (ale tylko jedna na plik) lub domyślnie - "wewnątrzpakietowo".
- Program rozpoczyna działanie od wywołania metody `public static void main(String args[])` z podanej klasy.

Strona główna

Strona tytułowa

Spis treści



Strona 21 z 80

Powrót

Full Screen

Zamknij

Koniec

Podstawy Javy na tle C++ 2

- Metody są implementowane w treści klasy.
- Obiekty są tworzone jedynie dynamicznie (poprzez **new**), ale nie trzeba ich dealokować - gdy przestają być używane, zwalnianiem pamięci zajmuje się tzw. *garbage collector*.
- Dziedziczenie jest publiczne i pojedyncze - mamy drzewo genealogiczne klas, którego korzeniem jest predefiniowany typ **Object**.
- Polimorfizm jest cechą podstawową metod - z założenia są "wirtualne". Za to dostęp do pól występujących w kilku zasłaniających się przez dziedziczenie wersjach realizowany jest podobnie, jak w C++ (nie ma czegoś takiego, jak "polimorfizm" dla pól).

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 22 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Pierwszy program

Przykład 1.

```
public class Hello {  
    public static void main(String args []) {  
        System.out.print(" Hello _World!\n" );  
    }  
}
```

Jak zwykle zaczynamy od programu wypisującego linijkę tekstu. Służy do tego metoda **print** statycznego obiektu klasy **PrintStream** o nazwie **out**. Obiekt ten jest umieszczony w klasie **System** standardowego środowiska języka. Wywołanie **print** znajduje się w metodzie **main** publicznej klasy **Hello**. Zauważmy tutaj, że plik w którym umieścimy ten programik musi nazywać się **Hello.java** zgodnie z formatem **<nazwa-klasy-publicznej>.java**. Domyślamy się więc, że w pojedynczym pliku może wystąpić tylko jedna klasa publiczna.

Strona główna

Strona tytułowa

Spis treści



Strona 23 z 80

Powrót

Full Screen

Zamknij

Koniec

Kompilacja i uruchamianie

Do kompilacji używamy programu `javac`
`javac Hello.java`

Powstaje plik `Hello.class`, który – uwaga – **nie** jest plikiem wykonywalnym. Ma on postać *B-kodu* i zawiera instrukcje zrozumiałe dla interpretera. Interpreter uruchamiamy poleceniem:
`java Hello`

zostanie wykonany zestaw instrukcji JVM zawarty w pliku `Hello.class`.

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 24 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Dokumentacja

Dokumentacja jawnego "biblioteki standardowej" - pakiety, hierarchia klas, nagłówki metod itd.

<http://java.sun.com/javase/6/docs/>

<http://java.sun.com/javase/6/docs/api/index.html>

Strona główna

Strona tytułowa

Spis treści

◀◀ ▶▶

◀ ▶

Strona 25 z 80

Powrót

Full Screen

Zamknij

Koniec

Typy i klasy

Typy podstawowe

Typy podstawowe zostały ujednoczone. Odpowiadają im opakowujące je klasy. Wartości logiczne (**true** - prawda, **false** - fałsz) przechowuje typ **boolean**.

Typ	Rozmiar	Min	Max	T. kopertowy
boolean	-	-	-	Boolean
char	16-bit	Unicode 0	Unicode $2^{16} - 1$	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-2^{15}	$+2^{15} - 1$	Short
int	32-bit	-2^{31}	$+2^{31} - 1$	Integer
long	64-bit	-2^{63}	$2^{63} - 1$	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double
void	-	-	-	Void

Strona główna

Strona tytułowa

Spis treści



Strona 26 z 80

Powrót

Full Screen

Zamknij

Koniec

Obiekty w Javie

W Javie nie ma wskaźników, ale obiekty tworzone są tylko dynamicznie (za pomocą **new** i odpowiedniego konstruktora klasy). Dostęp do nich możliwy jest jedynie przez referencje, które w przeciwieństwie do C++ nie są stałymi. Nie ma referencji na typy podstawowe - trzeba używać ich klas opakujących. Referencja pusta to **null**.

```
Hello obiekt = new Hello ();
```

```
...
```

```
obiekt.pole = ...; obiekt.metoda ();
```

```
...
```

```
Hello obiekt2;
```

```
obiekt2=obiekt; // to nie jest skopiowanie, dwie  
// referencje (jak wskaźniki w C) dotycza teraz  
// tego samego obiektu!
```

Od typu podstawowego lub klasy można utworzyć pochodny typ tablicowy, tablice są jednak obiektami i nie mają związku z żadnymi wskaźnikami (których w Javie brak). Nie można tworzyć klas potomnych dla tablic.

Strona główna

Strona tytułowa

Spis treści



Strona 27 z 80

Powrót

Full Screen

Zamknij

Koniec

Inicjalizacja zmiennych

W Javie inicjalizacja zmiennych jest obowiązkowa

```
void f() {  
    int i;  
    i++; // Bład – i not initialized  
}
```

```
public class InitialValues {  
    int i; // ok  
}
```

Uwaga: Pola klasy są inicjalizowane wartościami domyślnymi.

Strona główna

Strona tytułowa

Spis treści



Strona 28 z 80

Powrót

Full Screen

Zamknij

Koniec

Wartości domyślne dla typów podstawowych

- boolean: false
- char: '\u0000' czyli (char)0
- byte: 0
- short 0
- int 0
- long 0
- float 0.0
- double 0.0
- Object refrence: null;

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 29 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Inne typy liczbowe

- BigInteger - Liczby całkowite z nielimitowanego zakresu
- BigDecimal - Liczby niecałkowite z nielimitowanego zakresu, ale o ustalonej precyzji.

Strona główna

Strona tytułowa

Spis treści



Strona 30 z 80

Powrót

Full Screen

Zamknij

Koniec

Tworzenie obiektów

Przykład 2.

```
char c = 'x';  
Character C = new Character(c);
```

Lub z podobnym skutkiem:

```
Character C = new Character('x');
```

Uwaga: W przypadku typu `String` można pisać:

```
String s = "asdf";
```

zamiast

```
String s = new String("asdf");
```

Strona główna

Strona tytułowa

Spis treści



Strona 31 z 80

Powrót

Full Screen

Zamknij

Koniec

Klasa String

Wygodna klasa reprezentująca napisy, jej obiekty są jednak niemodyfikowalne. Liczne przeciążone konstruktory np.:

- `public String()` - tworzy napis pusty
- `public String(String s)` - konstruktor kopiujący

Użyteczne metody np.:

- `int length()` - długość napisu
- `char charAt(int)` - znak w napisie (na podanej pozycji)
- `int compareTo(String)` - podobna do funkcji `strcmp` z C
- `boolean equals(Object)` - sprawdza równość napisów

Zdefiniowano wiele operacji typowych dla napisów (np. złączanie `+ i +=`).

Klasy często definiują metodę ich konwersji na napis `public String toString()`, która jest używana np. przez `System.out.print` i `System.out.println`.

Strona główna

Strona tytułowa

Spis treści



Strona 32 z 80

Powrót

Full Screen

Zamknij

Koniec

Modyfikowanie napisów

Bardzo podobna, lecz modyfikowalna jest klasa `StringBuffer`
np. konkatencję

```
public static String x(String s)
{ return '<' +s + '>'; }
```

kompilator widzi jako

```
public static String x(String s)
{ return new StringBuffer().append('<')
.append(s).append('>').toString(); }
```

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



Strona 33 z 80

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Klasy opakowujące dla typów pierwotnych

Zawierają m. in.:

- konstruktor jednoargumentowy np. `Character(char)`
- konstruktor konwertujący z napisu (z wyjątkiem `Character`) np. `Integer(String)`
- metodę `String toString()` zwracającą napis reprezentujący wartość przechowywaną w obiekcie
- metodę dostępu do przechowywanej wartości np. `Character.charValue()`, `Boolean.booleanValue()`. Dla liczbowych typów pierwotnych dodatkowo metody te pozwalają na konwersję wartości między typami - klasy opakowujące mają pełen zestaw metod: `intValue()`, `longValue()`, `floatValue()`, `doubleValue()`.
- Wiele innych metod kodujących operacje charakterystyczne dla tych typów.

Strona główna

Strona tytułowa

Spis treści



Strona 34 z 80

Powrót

Full Screen

Zamknij

Koniec

Deklaracja i inicjalizacja tablic

Napisy:

```
int[] a1;
```

```
int a1[];
```

są równoważne i nie powodują przydziału pamięci na elementy tablicy - to tylko deklaracja.

Natomiast

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

```
int[] a = new int[5];
```

tworzy tablicę 5 elementową wraz z inicjalnymi wartościami.

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 35 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Przykład 3.

```
public class ArrayPrb {  
    public static void main(String [] args) {  
        int [] a1 = { 1, 2, 3, 4, 5 };  
        int [] a2;  
        a2 = a1;  
        for(int i = 0; i < a2.length; i++)  
            a2 [i]++;  
        for(int i = 0; i < a1.length; i++)  
            System.out.println(  
                "a1[" + i + "]=" + a1 [i]);  
    }  
}
```

wynik działania programu:

a1[0] = 2

a1[1] = 3

a1[2] = 4

a1[3] = 5

a1[4] = 6

Proszę zauważyć, że wartości w **a1** zostaną zwiększone.

Strona główna

Strona tytułowa

Spis treści



Strona 36 z 80

Powrót

Full Screen

Zamknij

Koniec

Operatory

Unarne

+ - ++ --

Arytmetyczne (i przesunięcia)

* / % + - << >> >>>

Relacyjne

> < >= <= == !=

Logiczne i bitowe

&& || & | ^

Warunkowe

A > B ? X : Y

Przypisania

=, *=, += <<= itp.)

Strona główna

Strona tytułowa

Spis treści



Strona 37 z 80

Powrót

Full Screen

Zamknij

Koniec

Szczegółowa tabela operatorów

1	[] () .	indeks tablicy wywołanie metody dostęp do pola	lewostronna
2	++ -- + - ~ !	inkrementacja dekrementacja plus i minus unarny NOT bitowe NOT logiczne	prawostronna
	(<typ>) new	rzutowanie typu tworzenie obiektu	
3	* / %	mnożenie dzielenie, reszta	lewostronna
4	+ - +	dodawanie, odejmowanie łączenie łańcuchów	lewostronna
5	<< >> >>>	przesunięcia bitowe	lewostronna
6	< <= > >= instanceof	porównania	lewostronna

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



Strona 38 z 80

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Szczegółowa tabela operatorów 2

7	==	równość	lewostronna
	!=	nierówność	
8	&	AND bitowe	lewostronna
9	^	XOR bitowe	lewostronna
10		OR bitowe	lewostronna
11	&&	AND logiczne	lewostronna
12		OR logiczne	lewostronna
13	? :	operator warunkowy	prawostronna
14	=	przypisanie	prawostronna
15	*= /= += -= %=	przypisania złożone	prawostronna

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



Strona 39 z 80

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Łączenie łańcuchów

Operator + może służyć do łączenia (konkatenacji) łańcuchów.

Przykład 4.

```
public class PRB2 {  
    public static void main(String args []) {  
        String S="list";  
        S=S+" _do";  
        S+=' _';  
        System.out.println(S + " cioci");  
        //list do cioci  
    }  
}
```

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 40 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Operator >>>

>>> realizuje przesunięcie bitowe uzupełniane zerami niezależnie od znaku (w C i C++ nie występuje). Dla przypomnienia operator >> uzupełnia liczby ujemne bitami ustawionymi na 1.

Przykład 5.

```
public class PRB2 {  
    public static void main(String args []) {  
        int i=-1;  
        i>>=1;  
        System.out.println(i); // -1  
        i=-1;  
        i>>>=1;  
        System.out.println(i); // 2147483647  
    }  
}
```

Strona główna

Strona tytułowa

Spis treści

◀◀ ▶▶

◀ ▶

Strona 41 z 80

Powrót

Full Screen

Zamknij

Koniec

Uwaga: Wykonanie operacji na typach prostych węższych niż `int`: `char`, `byte` lub `short` jest poprzedzone niejawną konwersją do typu `int`.

Przykład 6.

```
public class PRB1 {  
    public static void main(String args []) {  
        byte a;  
        a=100;  
        System.out.println(a*a); } //10000  
}
```

Strona główna

Strona tytułowa

Spis treści

◀◀ ▶▶

◀ ▶

Strona 42 z 80

Powrót

Full Screen

Zamknij

Koniec

Operacje arytmetyczne na typach kopertowych

Przykład 7.

```
public class ArithmeticPrb {  
    public static void main(String [] args) {  
        Integer i = new Integer (3);  
        Integer j;  
  
        j=i; // j nie jest nowym obiektem,  
            // a jedynie referencja do obiektu i  
        j=j+1; // wykonanie operacji arytmetycznej  
            // tworzy nowy obiekt  
        System.out.println(i + " + " + j);  
    }  
}
```

wynikiem działania tego programu będzie:

3 4

Strona główna

Strona tytułowa

Spis treści

◀◀ ▶▶

◀ ▶

Strona 43 z 80

Powrót

Full Screen

Zamknij

Koniec

Konkatenacja łańcuchów

Przykład 8.

```
public class StringConcatPrb {  
    public static void main(String [] args) {  
        String s1 = new String("Ala_ma");  
        String s2;  
  
        s2=s1; //s2 nie jest nowym obiektem,  
                //a jedynie referencja do obiektu s1  
        s1+="_kota"; //doklejenie lancucha znakow  
                    // tworzy nowy obiekt  
        System.out.println("s1 _:_ " + s1);  
        System.out.println("s2 _:_ " + s2);  
    }  
}
```

wynikiem działania tego programu będzie:

```
s1 : Ala ma kota  
s2 : Ala ma
```

Strona główna

Strona tytułowa

Spis treści

◀◀ ▶▶

◀ ▶

Strona 44 z 80

Powrót

Full Screen

Zamknij

Koniec

Wyrażenia, instrukcje, bloki

Deklaracje i kontrola sterowania

Każda deklaracja jest uważana za instrukcję.

Każda instrukcja kontroli sterowania jest instrukcją.

- pętla `for` - w Javie ma rozszerzoną składnię.
- pętla `while`
- pętla `do`
- `break`, `continue`, `return`
- `if`, `switch`

Uwaga: W języku angielskim funkcjonuje również pojęcie *instruction*, które odnosi się do instrukcji procesora, a w języku java ewentualnie do pojedynczej instrukcji kodu pośredniego.

Strona główna

Strona tytułowa

Spis treści



Strona 45 z 80

Powrót

Full Screen

Zamknij

Koniec

Sterowanie

Instrukcja if else

```
if(<wyrażenie>
  <instrukcja>
```

```
if(<wyrażenie>
  <instrukcja>
```

```
else
  <instrukcja>
```

Strona główna

Strona tytułowa

Spis treści



Strona 46 z 80

Powrót

Full Screen

Zamknij

Koniec

Pętla while

`while (<wyrażenie>) <instrukcja>`

Dopóki warunek po słowie **while** jest spełniony dopóty instrukcja lub *blok instrukcji* `<instrukcja>` będzie wykonywany. Trzeba zwrócić uwagę, aby pętla miała prawidłowy warunek końca i nie mogło zdarzyć się tak, iż będzie się ona wykonywać w nieskończoność.

W ogólności, problem stwierdzenia, czy program się zatrzyma, czy też nie, jest trudny. Popatrzmy na poniższy przykład.

Strona główna

Strona tytułowa

Spis treści



Strona 47 z 80

Powrót

Full Screen

Zamknij

Koniec

```

public class Collatz {
    static void collatz(int n) {
        while (n>1){
            System.out.print(n + " ");
            if (n % 2 == 0) n = (n / 2);
            else n=3*n+1;
        }
    }
    public static void main(String [] args) {
        int n = Integer.parseInt(args [0]);
        collatz(n);
        System.out.println ();
    }
}

```

Możemy zapytać, czy ta pętla się zawsze kiedyś zatrzyma. Jest to znany *problem Collatz'a*. Dotychczas nie udało się go rozwiązać, ani udowodnić, że jest nierozwiązywalny. Problem ten jest szczególnym przypadkiem *problemu stopu*, o którym wiadomo, że jest niealgorytmiczny (nie ma algorytmu, który go rozwiązuje).

[Strona główna](#)
[Strona tytułowa](#)
[Spis treści](#)

[Strona 48 z 80](#)
[Powrót](#)
[Full Screen](#)
[Zamknij](#)
[Koniec](#)

Pętla for

Pętla `for` często stosujemy do powtarzania pewnych operacji określoną liczbę razy, tak jak w przykładzie poniżej.

Przykład 9.

```
for(int i = 0; i < items1.length; i++)  
    menus[0].add(items1[i]);
```

Strona główna

Strona tytułowa

Spis treści



Strona 49 z 80

Powrót

Full Screen

Zamknij

Koniec

Pętla do while 2

do

<instrukcja>

while(<warunek kontynuowania petli >);

Zauważmy, że ta pętla musi się wykonać co najmniej raz.

Strona główna

Strona tytułowa

Spis treści



Strona 50 z 80

Powrót

Full Screen

Zamknij

Koniec

break i continue

- `break` - opuszcza pętlę
- `continue` - przechodzi do kolejnej iteracji

```
for (int i = 0; i < 100; i++) {  
    if (i == 74) break;  
    if (i % 9 != 0) continue;  
    System.out.println(i);  
}
```

Strona główna

Strona tytułowa

Spis treści

◀◀ ▶▶

◀ ▶

Strona 51 z 80

Powrót

Full Screen

Zamknij

Koniec

Instrukcja switch

```
switch(<selektor>) {  
  case <wartość> : <instrukcje>; break;  
  case <wartość> : <instrukcje>; break;  
  case <wartość> : <instrukcje>; break;  
  case <wartość> : <instrukcje>; break;  
  case <wartość> : <instrukcje>; break;  
  // ...  
  default: <instrukcje>;  
}
```

Strona główna

Strona tytułowa

Spis treści



Strona 52 z 80

Powrót

Full Screen

Zamknij

Koniec

```

public class VowelsAndConsonants {
    public static void main(String [] args) {
        char c = (char)(Math.random() * 26 + 'a');
        System.out.print(c + " :_");
        switch(c) {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u': System.out.println("vowel");
                    break;
            case 'y':
            case 'w': System.out.println("a_vowel_or_not");
                    break;
            default: System.out.println("consonant");
        }
    }
}

```

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 53 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

break, continue i etykiety

Możliwe jest opuszczenie za pomocą **break** nie tylko aktualnej, ale i bardziej "zewnątrznej" pętli, o ile opatrzy się ją etykietą.

```
<etykieta >:
<petla zewnetrzna > {
    <petla wewnetrzna > {
        //...
        break;
        //...
        continue;
        //...
        continue <etykieta >;
        //...
        break <etykieta >;
    }
}
```

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 54 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

break, continue i etykiety

```
public class LabeledWhile {
    public static void main(String [] args) {
        int i = 0;
        outer:
        while(true) {
            System.out.println("Outer_while_loop");
            while(true) {
                i++;
                System.out.println("i_=_ " + i);
                if(i==1){System.out.println("continue");
                    continue;}
                if(i==3){System.out.println("continue_outer");
                    continue outer;}
                if(i==5){System.out.println("break");
                    break;}
                if(i==7){System.out.println("break_outer");
                    break outer;}
            }
        }
    }
}
```

Strona główna

Strona tytułowa

Spis treści

◀◀ ▶▶

◀ ▶

Strona 55 z 80

Powrót

Full Screen

Zamknij

Koniec

break, continue i etykiety

Wyniki:

Outer **while** loop

i = 1

continue

i = 2

i = 3

continue outer

Outer **while** loop

i = 4

i = 5

break

Outer **while** loop

i = 6

i = 7

break outer

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 56 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Bloki

Blokiem nazywamy zero lub więcej instrukcji ujętych w nawiasy klamrowe. Blok ma swój zasięg.

Zasięg - czas życia zmiennych i obiektów

Przykład 10.

```
{  
    int x = 12;  
    { // nowy blok  
        int q = 96;  
        String s = new String("a string");  
    } //koniec zasięgu s,  
    //ale obiekt wskazywany przez s może "żyć" dalej  
} // koniec zasięgu zmiennej x
```

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



Strona 57 z 80

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Zmienne zagnieżdzone

Stosowanie *zmiennych zagnieżdzonych* o tej samej nazwie jak istniejąca zmienna w bloku zewnętrznym jest niedopuszczalne.

Przykład 11.

```
{  
    int x = 12;  
    {  
        int x = 96; // błąd  
    }  
}
```

Strona główna

Strona tytułowa

Spis treści



Strona 58 z 80

Powrót

Full Screen

Zamknij

Koniec

Nazwy

Nazwy w języku java nadajemy między innymi klasom, metodom i zmiennym. Każda nazwa musi spełniać pewne reguły aby była nazwą legalną, ponadto istnieją pewne zwyczaje nadawania nazw co czyni programy bardziej czytelnymi.

- Nazwa składa się z ciągu liter, cyfr, znaków podkreślenia _ oraz znaków \$.
- Pierwszy znak nazwy nie może być cyfrą.
- Małe i wielkie są rozróżniane (Java jest ang. case sensitive).
- Nazwa nie może być słowem kluczowym.

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 59 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Nazwy - konwencje

- Zwyczajowo znak \$ nie jest używany.
- Zwyczajowo pierwsza litera nazwy zmiennej jest mała.
- Zwyczajowo, jeśli nazwa zmiennej składa się z kilku słów, to poza pierwszym słowem pozostałe piszemy z wielkiej litery (podobnie nazwy metod).
- Zwyczajowo, nazwy klas piszemy wielką literą.
- Zwyczajowo, nazwy stałych piszemy kapitalikami, a kolejne słowa odzielamy kreską podkreślenia. Jest to jedyny wypadek, gdy użycie znaku _ jest zalecane.

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 60 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Nazwy - przykłady

- `int rozmiar;`
- `int numerButa;`
- `class NiewielkaKlasaODługiejNazwie{}`
- `static final int LICZBA_DNI_W_ROKU = 365;`

Strona główna

Strona tytułowa

Spis treści



Strona 61 z 80

Powrót

Full Screen

Zamknij

Koniec

Literały

Literałem (ang. literal) nazywamy reprezentację wartości zapisaną w kodzie źródłowym programu.

Przykład 12.

```
boolean result = true;  
char capitalC = 'C';  
byte b = 100;  
double d1 = 123.4;  
double d2 = 1.234e2; – to samo, co powyżej  
int octVal = 032; – wartość ósemkowa  
int hexVal = 0x1a; – wartość szesnastkowa
```

Uwaga: W javie literałem jest również **null** – wartość oznaczająca referencję pustą.

Strona główna

Strona tytułowa

Spis treści



Strona 62 z 80

Powrót

Full Screen

Zamknij

Koniec

Literały znakowe i napisowe

Literały znakowe ujmujemy w pojedyncze apostrofy, literały napisowe w cudzysłów.

Przypomnijmy, że znaki są kodowane w standardzie **Unicode (UTF-16)**. Można stosować kody `'\u<numer>'` takie jak `'\u0108'`, `'\u00ED'` (numer podajemy jako liczbę czterocyfrową w systemie szesnastkowym).

Podobnie jak w **C** można stosować znaki specjalne:

- `\b` (cofnięcie),
- `\t` (tabulacja),
- `\n` (nowa linia),
- `\f` (nowa strona),
- `\r` (powrót karetki),
- `\”` (cudzysłów),
- `\’` (apostrof),
- `\\` (ukośnik).

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)

[◀](#) [▶](#)

[◀](#) [▶](#)

[Strona 63 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Tworzenie klas

Deklaracja klasy:

```
class <nazwa klasy> {  
    <ciało klasy>  
}
```

Uwaga: słowo kluczowe **class** może poprzedzić specyfikator **public**. W danym pliku źródłowym powinna być tylko jedna klasa publiczna - o takiej samej nazwie, co plik.

Strona główna

Strona tytułowa

Spis treści



Strona 64 z 80

Powrót

Full Screen

Zamknij

Koniec

Przykład 13.

```
class DataOnly {  
    int i;  
    float f;  
    boolean b;  
}
```

Przykład 14.

```
DataOnly d = new DataOnly();
```

Strona główna

Strona tytułowa

Spis treści



Strona 65 z 80

Powrót

Full Screen

Zamknij

Koniec

Zmienne i klasy

Zmienne deklarowane wewnątrz klas możemy podzielić na kilka kategorii:

- Zmienne deklarowane bezpośrednio w bloku definicji klasy, te nazywamy *zmiennymi klasy* (polami klasy lub właściwościami).
- Zmienne deklarowane wewnątrz bloków kodu, te nazywamy *zmiennymi lokalnymi*.
- Zmienne występujące przy deklaracji metod, te nazywamy *parametrami*.

Strona główna

Strona tytułowa

Spis treści



Strona 66 z 80

Powrót

Full Screen

Zamknij

Koniec

```

public class Geometra {
    public static void main(String [] args) {
        Punkt p=new Punkt(1,1);
        Punkt q=new Punkt(2,2);
        System.out.println("|p,q|=" + p.odleglosc(q));
        System.out.println("PI=" + Math.PI);
    }
}

```

```

class Punkt {
    protected int x = 0;
    protected int y = 0;
    public Punkt(int x, int y) {
        this.x = x; // w Javie this to referencja
        this.y = y;
    }
    public double odleglosc(Punkt p) {
        double p1,p2;
        p1 = (x-p.x) * (x-p.x);
        p2 = (y-p.y) * (y-p.y);
        return java.lang.Math.sqrt(p1 + p2);
    }
}

```

Strona główna

Strona tytułowa

Spis treści



Strona 67 z 80

Powrót

Full Screen

Zamknij

Koniec

Deklaracje – pola klasy

Deklaracja pola klasy składa się z trzech elementów:

1. zero lub więcej modyfikatorów (np. specyfikatora dostępu),
2. typu pola,
3. nazwy pola.

Deklaracje – metody

Deklaracja metody składa się z sześciu elementów:

1. zero lub więcej modyfikatorów,
2. typu zwracanej wartości (może być `void`),
3. nazwy,
4. listy parametrów – parametry podajemy oddzielone przecinkami i umieszczamy w nawiasach okrągłych,
5. listy zgłaszanych wyjątków (omówimy później) i
6. ciała metody – ciągu instrukcji w nawiasach klamrowych.

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 68 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Modyfikatory

Specyfikatory dostępu

- **public** – pole, metoda jest widoczna poza klasą, w której występuje.
- **private** – pole, metoda **nie** jest widoczna poza klasą, w której występuje.
- **protected** – widoczność pośrednia, analogicznie jak w C++.
- **brak** – pole, metoda jest widoczna dla kodu z jej pakietu.

Inne modyfikatory to m.in.: **abstract**, **final**, **native**, **static**, **synchronized**, **volatile**. Modyfikator **final** dla pola jest odpowiednikiem **const** z C++, a dla metody lub klasy oznacza ”wersję finalną” (nie możliwą do zmiany w hierarchii dziedziczenia).

Strona główna

Strona tytułowa

Spis treści



Strona 69 z 80

Powrót

Full Screen

Zamknij

Koniec

Sygnatury

Sygnaturą metody (ang. method signature) nazywamy jej nazwę i typy parametrów.

Przykładowo metoda

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

ma sygnaturę

```
main(String[])
```

Strona główna

Strona tytułowa

Spis treści



Strona 70 z 80

Powrót

Full Screen

Zamknij

Koniec

Przeciążanie metod

Możliwa jest definicja, w obrębie jednej klasy kilku metod o takich samych nazwach lecz różnych sygnaturach. Taką konstrukcję programistyczną nazywamy *przeciążaniem* (ang. overloading).

Uwaga:Nadmierne stosowanie tych samych nazw dla różnych metod może doprowadzić do zmniejszenia czytelności kodu.

Uwaga:Przeciążone metody muszą się różnić typami argumentów, różnica w typie zwracanej wartości jest niewystarczająca (typ wartości zwracanej nie należy do sygnatury).

Poniższe deklaracje nie są poprawne.

```
void f() {}  
int f() {}
```

Strona główna

Strona tytułowa

Spis treści



Strona 71 z 80

Powrót

Full Screen

Zamknij

Koniec

Konstruktory klas

W momencie tworzenia obiektu (wystąpienia klasy) wywoływany jest konstruktor klasy tworzonej. Konstruktor deklaruje się podobnie jak metody klasy, przy czym konstruktor nie zwraca wartości, nazwa konstruktora musi być identyczna z nazwą klasy.

Uwaga: Konstruktory można przeciążać.

Uwaga: W przypadku braku deklaracji konstruktora kompilator dostarcza domyślny konstruktor bezargumentowy.

Przykład 15.

```
public Punkt(int x, int y) {  
    this.x = x;  
    this.y = y;  
}  
public Punkt() {  
    x = 0;  
    y = 0;  
}
```

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 72 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Inicjowanie pól obiektu

Przykład 16.

```
class Klasa {  
    int a=1,b;  
    final int c=4;  
    final int d;  
    public Klasa() { a=d=0; }  
    // public Klasa() { } - blad!  
}
```

Z chwilą tworzenia obiektu polem zdefiniowanym z podaniem wartości inicjalnej nadawane są te wartości, zaś pozostałym wartości domyślne zgodnie z ich typami. Następnie wywoływany jest konstruktor, który może zmodyfikować te przypisania.

Pól niemodyfikowalnych (z **final**) nie trzeba inicjować w definicji, ale wtedy każdy konstruktor musi nadać wartość takiemu polu, w przeciwnym razie wystąpi błąd kompilacji.

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 73 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Przekazywanie parametrów do metod

Przekazywanie typów prostych do metody odbywa się zawsze przez wartość.

Przykład 17.

```
public class ArgPassing {  
    public static void prb(int x) {  
        x++;  
        System.out.print(x + " ");  
    }  
    public static void main(String [] args) {  
        int x=0;  
        prb(x);  
        System.out.println(x);  
    }  
}
```

Wynikiem działania programu jest

1 0

Strona główna

Strona tytułowa

Spis treści

◀

▶

◀

▶

Strona 74 z 80

Powrót

Full Screen

Zamknij

Koniec

Przekazywanie parametrów do metod 2

Przekazywanie obiektów - zawsze przez referencję.

Przykład 18.

```
public class ArgPassing2 {  
    public static void prb(Punkt p) {  
        p.x++;  
        System.out.println("(" + p.x + ", " + p.y + ")");  
    }  
    public static void main(String [] args) {  
        Punkt p= new Punkt(0, 0);  
        prb(p);  
        System.out.println("(" + p.x + ", " + p.y + ")");  
    }  
}
```

Wynikiem działania programu jest

(1,0)

(1,0)

Strona główna

Strona tytułowa

Spis treści

◀ ▶

◀ ▶

Strona 75 z 80

Powrót

Full Screen

Zamknij

Koniec

Wartości zwracane

Metoda zwraca wartość do miejsca wywołania, gdy

- wszystkie instrukcje zostaną wykonane,
- wykonana zostanie instrukcja **return**,
- zostanie zgłoszony wyjątek (omówimy później).

Strona główna

Strona tytułowa

Spis treści

◀▶

◀▶

Strona 76 z 80

Powrót

Full Screen

Zamknij

Koniec

Słowo kluczowe `this`

W Javie jest to referencja – określona wewnątrz metody, pokazuje obiekt, na rzecz którego została ona wywołana.

Przykład 19.

```
class Punkt2 {  
    public int x = 0;  
    public int y = 0;  
    public Punkt2(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
class Punkt3 {  
    public int x = 0;  
    public int y = 0;  
    public Punkt3(int a, int b) {  
        x = a;  
        y = b;  
    }  
}
```

Strona główna

Strona tytułowa

Spis treści



Strona 77 z 80

Powrót

Full Screen

Zamknij

Koniec

Porównywanie obiektów

Przykład 20.

```
public class Equivalence {  
    public static void main(String [] args) {  
        Integer n1 = new Integer(47);  
        Integer n2 = new Integer(47);  
        System.out.println(n1 == n2); //false ,  
        System.out.println(n1 != n2); //true  
    }  
}
```

Porównane zostały referencje do obiektów (**Integer** jest klasą kopertową typu prostego **int**), a ponieważ są to dwa różne obiekty otrzymamy wynik taki jak w komentarzu. Bowiem porównywanie referencji sprawdza, czy pokazują one ten sam obiekt, ich zawartość nie jest badana!

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 78 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)

Porównywanie obiektów 2

Chcąc porównać zawartość korzystamy z metody `equals(Object)`, zdefiniowanej jeszcze w klasie `Object` (należy ją pokryć we własnych klasach).

Przykład 21.

```
public class EqualsMethod {  
    public static void main(String [] args) {  
        Integer n1 = new Integer(42);  
        Integer n2 = new Integer(42);  
        System.out.println(n1.equals(n2)); //true  
    }  
}
```

Strona główna

Strona tytułowa

Spis treści

◀◀ ▶▶

◀ ▶

Strona 79 z 80

Powrót

Full Screen

Zamknij

Koniec

Słowo kluczowe Static

```
class Punkt4 {  
    public int x = 0;  
    public int y = 0;  
    public static int ilePunktow=0;  
    public Punkt4(int x, int y) {  
        this.x = x;  
        this.y = y;  
        ilePunktow++;  
    }  
    public static void main(String [] args) {  
        Punkt4 p = new Punkt4(1, 1);  
        Punkt4 q = new Punkt4(0, 0);  
        System.out.println(p.ilePunktow);  
        System.out.println(q.ilePunktow);  
    }  
}
```

Wynikiem działania programu jest:

2
2

[Strona główna](#)

[Strona tytułowa](#)

[Spis treści](#)



[Strona 80 z 80](#)

[Powrót](#)

[Full Screen](#)

[Zamknij](#)

[Koniec](#)