Rozdział 2 Konstrukcja kodu

2.1 Metody konstrukcji kodu programów
2.1.1 Literały
2.1.2 Identyfikatory i słowa kluczowe.
2.1.3 Zmienne
2.1.4 Typy danych
2.1.5 Operatory
2.1.6 Instrukcje sterujące
2.2 Informacja praktyczna – wyświetlanie polskich znaków w konsoli platformy Javy

2.1 Metody konstrukcji kodu programów

Zapoznanie się z ogólną metodologią tworzenia programów w Javie pozwala na wejście w szczegółowy opis języka i tworzenie przykładowych aplikacji. Warto jednak najpierw określić szczegółowe zasady pisania kodu oraz stosowania komentarzy.

Czytelne pisanie kodu programu ułatwia w znaczny sposób unikania błędów oraz upraszcza przyswajalność kodu. Podstawą tworzenia kodu w sposób czytelny jest grupowanie kodu w bloki, wyróżnialne wcięciami. Blokiem nazwiemy zestaw kodu ujęty w nawiasy klamrowe:
np.:

```
instrukcja_grupująca{
        kod..
        kod..
} // koniec instrukcja_grupująca
```

Instrukcja grupująca rozpoczyna blok kodu. Instrukcją grupującą może być klasa, metoda, instrukcja warunkowa, instrukcja pętli, itp. Klamra otwierająca blok znajduje się bezpośrednio po instrukcji grupującej. Pierwsza fragment kodu po klamrze otwierającej umieszcza się w nowej linii. Każda linia bloku jest przesunięta względem pierwszego znaku instrukcji_grupującej o stałą liczbę znaków. Blok kończy się umieszczeniem klamry zamykającej w nowej linii na wysokości pierwszego znaku instrukcji grupującej. Blok warto kończyć komentarzem umożliwiającym identyfikację bloku zamykanego przez daną klamrę.
W konstrukcji programu należy pamiętać, że język Java rozróżnia małe i wielkie litery, Oznacza to, że przykładowe nazwy zmiennych „Jedi” oraz „jedi” nie są równoważne i określają dwie różne zmienne. Dodatkowe elementy stylu programowania, o których należy pamiętać to:
- stosowanie nadmiarowości celem poprawienia czytelności kodu np.:
*int a=(c * 2) + (b / 3);* zamiast *int a=c* 2 + b/ 3;*
- stosowanie separatorów np.:
*int a=(c * 2) + (b / 3);* zamiast *int a=(c*2)+(b/3);*
- konsekwencja w oznaczaniu, np.:
int[] n;
int n[];
definicja jednoznaczna lecz różna forma.

Ważnym elementem czytelnej konstrukcji kodu jest używanie dokumentacji kodu za pomocą komentarzy. W Javie stosuje się trzy podstawowe typy komentarza:
- komentarz liniowy:
// miejsce na komentarz do końca linii
- komentarz blokowy:
/*
miejsce na komentarz w dowolnym miejscu bloku
*/
- komentarz dokumentacyjny
/**
miejsce na komentarz w dowolnym miejscu bloku
*/

Komentarz liniowy wykorzystuje się do krótkiego oznaczania kodu np. interpretacji zmiennych, oznaczania końca bloku, itp. Komentarz blokowy stosuje się do chwilowego wyłączania kodu z kompilacji oraz do wprowadzania szerszych opisów kodów. Komentarz dokumentacyjny używa się do tworzenia dokumentacji kodu polegającej na opisie klas i metod, ich argumentów, dziedziczenia, twórców kodu, itp. Komentarz dokumentacyjny w Javie może zawierać dodatkowe elementy takie jak:
@author - umożliwia podanie autora kodu,
@version - umożliwia podanie wersji kodu,
@see - umożliwia stworzenie referencji,
itp.
Pełny zestaw elementów formatujących znajduje się w opisie narzędzi *javadoc*, tworzących dokumentację na podstawie kodu i komentarza dokumentacyjnego. Należy pamiętać, że standardowo zostaną opisane tylko elementy kodu oznaczane jako *public* i *protected*.
O czym należy pamiętać używając komentarzy w Javie? Otóż o tym, że komentarz jest zastępowany spacją, co oznacza że kod:

*double pi = 3.14/*kto by to wszystko zapamiętał*/56;*

jest interpretowany jako *double pi = 3.14 56;* co jest błędem.

Przykład 2.1

```
// Jedi3.java :

/**
RycerzJedi określa klasę rycerzy Jedi
@author Jacek Rumiński
@version 1.0
*/
class RycerzJedi{
        /**     nazwa - określa nazwę rycerza Jedi */
        String nazwa;
        /**     kolor_miecza - określa kolor miecza  rycerza Jedi */
        String kolor_miecza;

        /**     konstruktor umożliwia podanie właściwości obiektu */
        RycerzJedi(String nazwa, String kolor_miecza){
                this.nazwa=nazwa;
```

```
                this.kolor_miecza=kolor_miecza;
        }

        /**     metoda opis wyświetla zestaw właściwości rycerza Jedi */
        void opis(){
                System.out.println("Rycerz "+nazwa+ " ma "+kolor_miecza+" miecz.");
        }

}// koniec class RycerzJedi
/**
Jedi3 określa klasę główna aplikacji
@author Jacek Rumiński
@version 1.0
*/
public class Jedi3{

        public static void main(String args[]){
                RycerzJedi luke = new RycerzJedi("Luke", "niebieski");
                RycerzJedi ben = new RycerzJedi("Obi-wan","biały");
                luke.opis();
                ben.opis();
        }// koniec public static void main(String args[])

}// koniec public class Jedi3
```

Tworząc dokumentację HTML dla kodu z przykładu 2.1 należy w następujący sposób wykorzystać narzędzie *javadoc*:

javadoc -private -author -version Jedi3.java

gdzie: -*private* powoduje, że wygenerowana zostanie dokumentacja dla wszystkich typów elementów private, protected i public;
-*author* powoduje, że wygenerowana zostanie informacja o autorze,
-*version* powoduje, że wygenerowana zostanie informacja o wersji.

W rezultacie otrzymuje się szereg stron w formacie HTML, wraz ze stroną startową index.html.

2.1.1 Literały

Literały oznaczają nazwy zmiennych, których typ oraz wartości wynikają z zapisu. Przykładowo „Niech moc będzie z Wami!” jest literałem o typie łańcuch znaków (String) i wartości *Niech moc będzie z Wami!*. Inne przykłady to: true, false, 12, 0.1, ‘e’, 234L, 0.34f, itp. Tłumaczenie kodu z literałami jest pobieraniem konkretnych wartości typu łańcuch znaków, znak, liczba rzeczywista, liczba całkowita, wartość logiczna (true lub false). Uwaga, wartości logiczne nie są jednoznaczne w Javie z wartościami liczbowymi np. typu 0 i 1.

2.1.2 Identyfikatory i słowa kluczowe.

Stworzenie programu wymaga znajomości zasad tworzenia identyfikatorów oraz podstawowych słów kluczowych. Zasady tworzenia identyfikatorów określają reguły konstrukcji nazw pakietów, klas, interfejsów, metod i zmiennych. Identyfikatory są więc elementami odwołującymi się do podstawowych elementów języka. Identyfikator tworzy się korzystając z liter, liczb, znaku podkreślenia „_” oraz znaku „$”. Tworzona nazwa nie może się jednak zaczynać liczbą. Pierwszym znakiem może być litera bądź symbol podkreślenia „_” lub „$”. W Javie rozróżnialne są wielkie i małe litery w nazwach, tak więc nazwy: *Jedi* i *jedi* oznaczać będą dwa różne identyfikatory!

Słowa kluczowe to identyfikatory o specjalnym znaczeniu dla języka Java. Identyfikatory te są już zdefiniowane dla języka i posiadają określone znaczenie w kodzie programu. Wszystkie więc nazwy w kodzie źródłowym będące poza komentarzem i i nie są stworzonymi przez programistę identyfikatorami są traktowane jako słowa kluczowe i ich znaczenie jest poszukiwane przez kompilator w czasie kompilacji. Słowa kluczowe (nazwy) są więc zarezerwowane dla języka i programista nie może używać tych nazw dla konstrukcji własnych identyfikatorów. Następujące słowa kluczowe są zdefiniowane w Javie:

abstract - deklaruje klase lub metodę jako abstrakcyjną,
boolean - deklaruje zmienną lub wartość zwracaną jako wartość logiczną,
break - przerwanie,
byte - deklaruje zmienną lub wartość zwracaną jako wartość byte,
case - określa opcję w instrukcji wyboru switch,
catch - obsługuje wyjątek,
char - deklaruje zmienną lub wartość zwracaną jako wartość znaku,
class - oznacza początek definicji klasy,
const,
continue - przerywa pętle rozpoczynając jej kolejny cykl,
default - określa domyślne działąnie dla instrukcji wyboru switch,
do - rozpoczyna blok instrukcji w pętli while,
double - deklaruje zmienną lub wartość zwracaną jako wartość rzeczywistą o podwójnej precyzji,
else - określa kod warunkowy do wykonania jeśli nie jest spełniony warunek w instrukcji if,
extends - określa, że definiowana klasa dziedziczy z innej,
final - określa pole, metodę lub klasę jako stałą,
finally - gwarantuje wykonywalność blok kodu,
float - deklaruje zmienną lub wartość zwracaną jako wartość rzeczywistą,
for - określa początek pętli for,
goto,
if - określa początek instrukcji warunkowej if,
implements - deklaruje, że klasa korzysta z danego interfejsu,
import - określa dostęp do pakietów i klas,
instanceof - sprawdza czy dany obiekt jest wystąpieniem określonej klasy,
int - deklaruje zmienną lub wartość zwracaną jako wartość całkowitą 4 bajtową,
interface - określa początek definicji interfejsu,
long - deklaruje zmienną lub wartość zwracaną jako wartość całkowitą 8 bajtową,
native - określa, że metoda jest tworzona w kodzie danej platformy (np. C),
new - wywołuje nowy obiekt,

package - określa nazwę pakietu do którego należy dana klasa,
private - deklaruje metodę lub pole jako prywatne,
protected - deklaruje metodę lub pole jako chronione,
public - deklaruje metodę lub pole jako dostępne,
return - określa zwracanie wartości metody,
short - deklaruje zmienną lub wartość zwracaną jako wartość całkowitą 2 bajtową,
static - deklaruje pole, metodę lub kod jako statyczny,
super - odniesienie do rodzica danego obiektu,
switch - początek instrukcji wyboru,
synchronized - oznacza fragment kodu jako synchronizowany,
this - odniesienie do aktualnego obiektu,
throw - zgłasza wyjątek,
throws - deklaruje wyjątek zgłaszany przez metodę,
transient - oznacza blokadę pola przed szeregowaniem,
try - początek bloku kodu, który może wygenerować wyjątek,
void - deklaruje, że metoda nie zwraca żadnej wartości,
volatile - ostrzega kompilator, że zmienna może się zmieniać asynchronicznie,
while - początek pętli.

przy czym słowa kluczowe *const* (stała) oraz *goto* (skok do) są zarezerwowane lecz nie używane.

2.1.3 Zmienne

Przetwarzanie informacji w kodzie programu wymaga zdefiniowania i pracy ze zmiennymi. Zmienne są to elementy programu wynikowego reprezentowane w kodzie źródłowym przez identyfikatory, literały i wyrażenia. Zmienne są zawsze określonego typu. Typ zmiennych (typ danych) definiowany jest w Javie przez typy podstawowe i odnośnikowe (referencyjne). W celu pracy ze zmiennymi należy zapoznać się z możliwymi typami danych oferowanymi w środowisku Java.

2.1.4 Typy danych

Podstawowe typy danych definiowane w specyfikacji Javy charakteryzują się następującymi właściwościami:
1. Typy danych są niezależne od platformy sprzętowej (w C i C++ często inaczej interpretowane były zmienne zadeklarowane jako dany typ, np. dla zmiennej typu *int* możliwe były reprezentacje w zależności od platformy 2 lub 4 bajtowe. Konieczne było stosowanie operatora *sizeof* w celu uzyskania informacji o rozmiarze zmiennej) i ich rozmiar jest stały określony w specyfikacji.
2. Konwersja (*casting* - rzutowanie) typów danych jest ograniczona tylko dla liczb. Nie można dokonać więc konwersji wprost pomiędzy typem znakowym a liczbą, czy pomiędzy typem logicznym a liczbą.
3. Wszystkie typy liczbowe są przechowywane za znakiem, np. *byte*: -128..0..127. Nie ma typów oznaczanych w innych językach jako *unsigned*, czyli bez znaku.
4. Wszystkie podstawowe typy danych są oznaczane z małych liter.
5. Nie istnieje w gronie podstawowych typów danych typ łańcucha znaków.
6. Istnieją klasy typów danych oznaczanych z wielkich liter, umożliwiające konwersję i inne operacje na obiektach tych klas, a przez to na podstawowych typach danych.

Wśród licznych klas typów danych znajduje się klasa *String*, umożliwiająca tworzenie obiektów reprezentujących łańcuch znaków. Typ łańcucha znaków jest więc tylko typem referencyjnym, gdyż odnosi się do klasy, a nie do podstawowego typu danych.

Następujące podstawowe typy danych są zdefiniowane w specyfikacji Javy:

boolean : (1 bit) typ jednobitowy oznaczający albo true albo false. Nie może podlegać konwersji do postaci liczbowej, czyli nie możliwe jest znaczenie typu jako 1 lub 0. Oznaczanie wartości typów (jak wszystkie w Javie) jest ściśle związane z wielkością liter. Przykładowe oznaczenia TRUE czy False nie mają nic wspólnego z wartościami typu *boolean*.

byte : (1 bajt) typ liczbowy, jednobajtowy za znakiem. Wartości tego typu mieszczą się w przedziale: -128 do 127.

short : (2 bajty) typ liczbowy, dwubajtowy ze znakiem. Wartości tego typu mieszczą się w przedziale: -32,768 do 32,767

int : (4 bajty) typ liczbowy, czterobajtowy ze znakiem. Wartości tego typu mieszczą się w przedziale: -2,147,483,648 do 2,147,483,647.

long : (8 bajtów) typ liczbowy, ośmiobajtowy ze znakiem. Wartości tego typu mieszczą się w przedziale: -9,223,372,036,854,775,808 do +9,223,372,036,854,775,807.

float : (4 bajty, spec. IEEE 754) typ liczb rzeczywistych, czterobajtowy ze znakiem. Wartości tego typu mieszczą się w przedziale: 1.40129846432481707e-45 to 3.40282346638528860e+38 (dodatnie lub ujemne).

double : (8 bajtów spec. IEEE 754) typ liczb rzeczywistych, ośmiobajtowy ze znakiem. Wartości tego typu mieszczą się w przedziale: 4.94065645841246544e-324d to 1.79769313486231570e+308d (dodatnie lub ujemne).

char : (2 bajty), typ znakowy dwubajtowy, dodatni. Kod dwubajtowy umożliwia zapis wszelkich kodów w systemie Unicode, który to jest standardem w Javie. Zakres wartości kodu to: 0 to 65,535. Tablica znaków nie jest łańcuchem znaków, tak jak to jest interpretowane w C czy C++.

void: typ nie jest reprezentowany przez żadną wartość, wskazuje, że dana metoda nic nie zwraca.

W Javie w bibliotece podstawowej języka: *java.lang.** znajdują się następujące klasy typów danych:

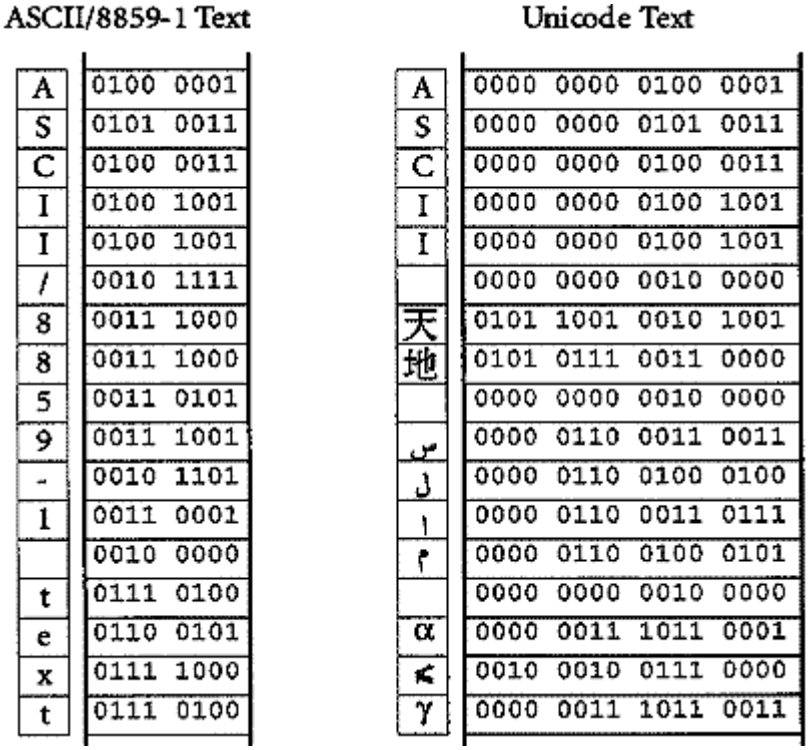
Boolean: klasa umożliwiająca stworzenie obiektu przechowującego pole o wartości typu podstawowego *boolean*. Klasa ta daje liczne możliwości przetwarzania wartości typu *boolean* na inne. Możliwe jest przykładowo uzyskanie reprezentacji znakowej (łańcuch znaków -> obiekt klasy *String*). Jest to możliwe poprzez wywołanie metody *toString()*. Metoda ta jest stosowana dla wszystkich klas typów danych, tak więc

możliwa jest konwersja (rzutowanie) dowolnej liczby na łańcuch znaków. Istnieje również statyczna metoda klasy *Boolean* (nie trzeb tworzyć obiektu aby się do metody statycznej odwoływać) zwracająca wartość typu podstawowego boolean na podstawie argumentu metody będącego łańcuchem znaków, np. *Boolean.valueof(„yes”)*; zwraca wartość *true*;

Byte : klasa umożliwiająca stworzenie obiektu przechowującego pole o wartości typu podstawowego *byte.* Klasa ta umożliwia liczne konwersje typu liczbowego byte do innych podstawowych typów liczbowych. Stosowane są przykładowo następujące metody tej klasy: intValue() - konwersja wartości typu *byte* na typ *int*; floatValue()- konwersja wartości typu *byte* na typ *float*; longValue()- konwersja wartości typu *byte* na typ *long*; i inne. Statyczne pola tej klasy: MIN_VALUE oraz MAX_VALUE, umożliwiają pozyskanie rozmiaru danego typu w Javie. Podobne właściwości posiadają inne klasy liczbowych typów danych:

Short ,
Integer,
Long ,
Float ,
Double .

Character : klasa umożliwiająca stworzenie obiektu przechowującego pole o wartości typu podstawowego *char.* Zdefiniowano obszerny zbiór pól statycznych oraz metod tej klasy. Większość pól i metod dotyczy obsługi standardowej strony kodowej platformy Javy czyli Unicodu (wrersja 2.0). Programy w Javie są zapisywane w Unicodzie. Unicode jest systemem reprezentacji znaków graficznych poprzez liczby z zakresu 0-65,535. Pierwsze 128 znaków kodu Unicode to znaki kodu ASCII (ANSI X3.4) czyli American Standard Code for Information Interchange. Kolejne 128 znaków w Unicodzie to odpowiednio kolejne 128 znaków w rozszerzonym kodzie ASCII (ISO Latin 1). Z wyjątkiem komentarzy, identyfikatorów i literałów znakowych i łańcuchów znaków wszelki kod w Javie musi być sformowany przez znaki ASCII (bezpośrednio lub jako kody Unicodu zaczynające się od sekwencji ‘\u’, np. ‘\u00F8’). Oznacza to, że np. komentarz może być zapisany w Unicodzie.



Rysunek 1. Porównanie kodowania znaków w ASCII i Unicodzie 2.0.

Wiele jednak programów stworzonych w Javie wymaga przetwarzania danych tekstowych z wykorzystaniem innych systemów kodowania znaków, np. ISO-8859-2. Konieczne są więc mechanizmy konwersji standardów kodowania. Mechanizmy takie są dostępne w Javie. Następujące standardy kodowania (strony kodowe) są obsługiwane przez Javę:

Nazwa strony kodowej:	***Opis:***
ASCII:	ASCII
ISO8859_1 :	ISO 8859-1
ISO8859_2:	**ISO 8859-2**
ISO8859_3:	ISO 8859-3
ISO8859_4:	ISO 8859-4
ISO8859_5:	ISO 8859-5
ISO8859_6:	ISO 8859-6
ISO8859_7:	ISO 8859-7
ISO8859_8:	ISO 8859-8
ISO8859_9:	ISO 8859-9
ISO8859_15_FDIS:	ISO 8859-15 (Final Draft Information Standard, based on 8859-1)
Big5:	Big5, Traditional Chinese
Cp037:	USA, Canada(Bilingual, French), Netherlands, Portugal, Brazil, Australia
Cp1006:	IBM AIX Pakistan (Urdu)
Cp1025:	IBM Multilingual Cyrillic: Bulgaria, Bosnia, Herzegovinia, Macedonia(FYR)

Cp1026:	IBM Latin-5, Turkey
Cp1046:	IBM Open Edition US EBCDIC
Cp1097:	IBM Iran(Farsi)/Persian
Cp1098:	IBM Iran(Farsi)/Persian (PC)
Cp1112:	IBM Latvia, Lithuania
Cp1122:	IBM Estonia
Cp1123:	IBM Ukraine
Cp1124:	IBM AIX Ukraine
Cp1140:	Cp037 with the euro
Cp1141:	Cp273 with the euro
Cp1142:	Cp277 with the euro
Cp1143:	Cp278 with the euro
Cp1144:	Cp280 with the euro
Cp1145:	Cp284 with the euro
Cp1146:	Cp285 with the euro
Cp1147:	Cp297 with the euro
Cp1148:	Cp500 with the euro
Cp1149:	Cp871 with the euro
Cp1250:	**Windows Eastern European**
Cp1251:	Windows Cyrillic
Cp1252:	Windows Latin-1
Cp1253:	Windows Greek
Cp1254:	Windows Turkish
Cp1255:	Windows Hebrew
Cp1256:	Windows Arabic
Cp1257:	Windows Baltic
Cp1258:	Windows Vietnamese
Cp1381:	IBM OS/2, DOS People's Republic of China (PRC)
Cp1383:	IBM AIX People's Republic of China (PRC)
Cp273:	IBM Austria, Germany
Cp277:	IBM Denmark, Norway
Cp278:	IBM Finland, Sweden
Cp280:	IBM Italy
Cp284:	IBM Catalan/Spain, Spanish Latin America
Cp285:	IBM United Kingdom, Ireland
Cp297:	IBM France
Cp33722:	IBM-eucJP - Japanese (superset of 5050)
Cp420:	IBM Arabic
Cp424:	IBM Hebrew
Cp437:	MS-DOS United States, Australia, New Zealand, South Africa
Cp500:	EBCDIC 500V1
Cp737:	PC Greek
Cp775:	PC Baltic
Cp838:	IBM Thailand extended SBCS
Cp850:	MS-DOS Latin-1
Cp852:	**MS-DOS Latin-2**
Cp855:	IBM Cyrillic
Cp857:	IBM Turkish
Cp858:	Cp850 with the euro
Cp860:	MS-DOS Portuguese

Cp861: MS-DOS Icelandic
Cp862: PC Hebrew
Cp863: MS-DOS Canadian French
Cp864: PC Arabic
Cp865: MS-DOS Nordic
Cp866: MS-DOS Russian
Cp868: MS-DOS Pakistan
Cp869: IBM Modern Greek
Cp870: IBM Multilingual Latin-2
Cp871: IBM Iceland
Cp874: IBM Thai
Cp875: IBM Greek
Cp918: IBM Pakistan(Urdu)
Cp921: IBM Latvia, Lithuania (AIX, DOS)
Cp922: IBM Estonia (AIX, DOS)
Cp930: Japanese Katakana-Kanji mixed with 4370 UDC, superset of 5026
Cp933: Korean Mixed with 1880 UDC, superset of 5029
Cp935: Simplified Chinese Host mixed with 1880 UDC, superset of 5031
Cp937: Traditional Chinese Host miexed with 6204 UDC, superset of 5033
Cp939: Japanese Latin Kanji mixed with 4370 UDC, superset of 5035
Cp942: Japanese (OS/2) superset of 932
Cp948: OS/2 Chinese (Taiwan) superset of 938
Cp949: PC Korean
Cp950: PC Chinese (Hong Kong, Taiwan)
Cp964: AIX Chinese (Taiwan)
Cp970: AIX Korean
EUC_CN: GB2312, EUC encoding, Simplified Chinese
EUC_JP: JIS0201, 0208, 0212, EUC Encoding, Japanese
EUC_KR: KS C 5601, EUC Encoding, Korean
EUC_TW: CNS11643 (Plane 1-3), T. Chinese, EUC encoding
GBK: GBK, Simplified Chinese
ISO2022CN: ISO 2022 CN, Chinese
ISO2022CN_CNS: CNS 11643 in ISO-2022-CN form, T. Chinese
ISO2022CN_GB: GB 2312 in ISO-2022-CN form, S. Chinese
ISO2022JP: JIS0201, 0208, 0212, ISO2022 Encoding, Japanese
ISO2022KR: ISO 2022 KR, Korean
JIS0201: JIS 0201, Japanese
JIS0208: JIS 0208, Japanese
JIS0212: JIS 0212, Japanese
KOI8_R: KOI8-R, Russian
MS874: Windows Thai
MacArabic: Macintosh Arabic
MacCentralEurope: Macintosh Latin-2
MacCroatian: Macintosh Croatian
MacCyrillic: Macintosh Cyrillic
MacDingbat: Macintosh Dingbat
MacGreek: Macintosh Greek
MacHebrew: Macintosh Hebrew

MacIceland:	Macintosh Iceland
MacRoman:	Macintosh Roman
MacRomania:	Macintosh Romania
MacSymbol:	Macintosh Symbol
MacThai:	Macintosh Thai
MacTurkish:	Macintosh Turkish
MacUkraine:	Macintosh Ukraine
SJIS:	Shift-JIS, Japanese
UTF8:	UTF-8

Metody dostępne w klasie Character umożliwiają ponadto przetwarzanie znaków i ich identyfikację. Przykładowo za pomocą metody *isWhitespace (char ch)* można uzyskać informację czy dany znak jest znakiem sterującym zwanym jako :"Java whitespace", do których zalicza się:
- Unicode: spacja (category "Zs"), lecz nie spacja oznaczana przez (\u00A0 lub \uFEFF).
- Unicode: separator linii (category "Zl").
- Unicode: separator paragrafu (category "Zp").
- \u0009, HORIZONTAL TABULATION.
- \u000A, LINE FEED.
- \u000B, VERTICAL TABULATION.
- \u000C, FORM FEED.
- \u000D, CARRIAGE RETURN.
- \u001C, FILE SEPARATOR.
- \u001D, GROUP SEPARATOR.
- \u001E, RECORD SEPARATOR.
- \u001F, UNIT SEPARATOR.

String - klasa umożliwiająca stworzenie nowego obiektu typu łańcuch znaków. Nie istnieje typ podstawowy łańcucha znaków, tak więc klasa ta jest podstawą tworzenia wszystkich zmiennych przechowujących tekst. Obiekt klasy String może być inicjowany następująco:

String str = „Rycerz Luke";
lub
String str = new String(„Rycerz Luke"); //i inne konstruktory klasy String.

Pierwszy sposób inicjowania przez referencje jest podobne do inicjowania zmiennych typów podstawowych. Drugi sposób jest inicjowania jest jawnym wywołaniem konstruktora klasy String. Klasa String umożliwia szereg operacji na łańcuchach znaków (np. zmiana wielkości, itp.). Należy pamiętać, że tablica znaków *char* nie jest rozumiana jako obiekt String. Obiekt klasy String może być stworzony za pomocą tablicy znaków, np.: String str = new String(c); gdzie c jest tablicą znaków np.: char c[] = new char[10]. Wśród konstruktorów klasy String znajduje się również konstruktor umożliwiający stworzenie łańcucha znaków na podstawie tablicy bajtów, według podanej strony kodowej, np.: String str = new String (b, „Cp1250"), gdzie b to tablica bajtów, np.: byte b[] = new byte[10];. Warto pamiętać również o tym, że obiekt klasy String nie reprezentuje sobą wartości typu podstawowego, stąd nie można porównywać dwóch łańcuchów znaków bezpośrednio, lecz poprzez metodę klasy

String: public boolean equals(Object anObject). Poniższy przykład obrazuję porównywanie łańcuchów znaków.

Przykład 2.2

```
//Moc.java

public class Moc{

	public static void main(String args[]){

		String dobro = new String("Dobro - jasna strona mocy");
		System.out.println("Ciemna strona mocy twierdzi:");
		String zlo = new String("Dobro - jasna strona mocy");

		if (zlo == dobro){
			System.out.println("Moc to jedno");
		} else
			System.out.println("Dwie moce?");

		if (zlo.equals(dobro)){
			System.out.println("Moc to jedno");
		} else
			System.out.println("Dwie moce?");
	}

}// koniec public class Moc
```

Object - klasa ta jest klasą nadrzędną wszystkich klas w Javie, tak więc tworzenie własnych typów danych będących klasami jest odwołaniem się do obiektu klasy Object.

Void - przechowuje referencje do obiektu klasy Class reprezentującej typ podstawowy *void*.

Poniżej zaprezentowano przykładową aplikację ukazującą różne typy danych podstawowych i klasy typów danych.

Przykład 2.3:

```
//Typy.java

public class Typy{

	public static void main(String args[]){
		boolean prawda[] = new boolean[2];
		prawda[0]=true;
		byte bajt = (byte)0;
		int liczba = 135;
		long gluga = 123456789L;
		char znak1 = '\u0104'; \\ w Unicodzie kod litery ‘Ą’
```

```
        char znak2 = 'ą';
        char znaki[] = { 'M', 'O', 'C'};
        System.out.println("Oto zmienne: ");
        System.out.println("boolean= "+prawda[1]);
        System.out.println("byte= "+bajt);
        System.out.println("int= "+liczba);
        System.out.println("char= "+znak1);
        System.out.println("char= "+znak2);

        Integer liczba1 = new Integer(liczba);
        bajt=liczba1.byteValue();
        System.out.println("byte= "+bajt);
        String str = new String(znaki);
        System.out.println(str);
    }

}// koniec public class Typy
```

Warto zauważyć, ze chociaż wszystkie zmienne muszą być zainicjowane jeżeli mają być użyte, to zmienna typu boolean *prawda* jest zadeklarowana jako tablica dwóch elementów, z czego drugi element nie jest inicjowany jawnie. Drugi element jest inicjowany w tablicy automatycznie na wartość domyślną.
Następujące wartości domyślne są przyjmowane dla zmiennych poszczególnych typów podstawowych:

boolean:	false
char:	'\u0000' (null)
byte:	(byte)0
short:	(short)0
int:	0
long:	0L
float:	0.0f
double:	0.0 (lub 0.0d)

Zmienna podstawowego typu jest konkretną wartością przechowywaną na stosie, co umożliwia efektywny do niej dostęp. Zmienna typu *Class* jest interpretowana jako „uchwyt" do obiektu typu Class, np.: String l; oznacza deklaracje uchwytu l do obiektu klasy String. Zadeklarowany uchwyt jest również przechowywane na stosie, nie mniej konieczne jest zainicjowanie uchwytu, lub inaczej wskazanie obiektu, który będzie związany z uchwytem. Przykładowo przypisanie obiektu do uchwytu może wyglądać następująco: l = new String („Jedi");. W wyniku wykonania instrukcji new powstaje obiekt danej klasy (np. String), który jest przechowywany na stercie. Przechowywanie wszystkich obiektów na stercie ma tę zaletę, że w czasie kompilacji nie jest potrzebna wiedza na temat wielkości pamięci jaka musi być zarezerwowana. Wadą przechowywania obiektów na stercie jest dłuższy czas dostępu do obszarów pamięci sterty, co powoduje spowolnienie pracy programu.

Oprócz istniejących typów podstawowych oraz klas typów zdefiniowanych w pakiecie java.lang.* (czyli np. Boolean, Integer, Byte, itp.) można skorzystać z tysięcy innych typów danych. Definicją typu jest definicja każdej klasy. Stąd też istnieje możliwość skorzystania z typów danych (klas) wbudowanych w biblioteki standardowe i rozszerzone języka Java lub można stworzyć własne typy danych (klasy).

2.1.5 Operatory

Operatory to elementy języka służące do generacji nowych wartości na podstawie podanych argumentów (jeden lub więcej). Operator wiąże się więc najczęściej z określonym działaniem na zmiennych. Prawie wszystkie operatory (z wyjątkiem: '=', '==', '!=', '+', '+=') działają na podstawowych typach danych, a nie na obiektach.
Wyróżnia się następujące klasy operatorów podane wedle ich kolejności wykonywania:
- operatory negacji;
- operatory matematyczne,
- operatory przesunięcia,
- operatory relacji,
- operatory logiczne i bitowe,
- operatory warunkowe,
- operatory przypisania.

Operator negacji powoduje zmianę wartości zmiennej na przeciwną pod względem znaku, np.: int a =4; x = -a; (to x jest równe -4), itd.

Operatory matematyczne to takie operatory, które służą do wykonywania operacji matematycznych na argumentach. Do operacji matematycznych zalicza się:
- mnożenie '*';
- dzielenie '/';
- modulo - reszta z dzielenia '%',
- dodawanie '+',
- odejmowanie '-'.

W wyniku dzielenia liczba całkowitych Java nie zaokrągla wyników do najbliższej wartości całkowitej, lecz obcina powstałą liczbę do liczby całkowitej. Dodatkowym elementem wykonywania operacji matematycznych w Javie (podobnie jak i w C) jest skrócony zapis operacji matematycznych jeśli jest wykonywana operacja na zmiennej, która przechowuje zarazem wynik tej operacji. Wówczas możliwe są następujące skrócone zapisy operacji:
- zwiększanie / zmniejszanie o 1 wartości zmiennej:

zapis klasyczny, np.: x = x+1; x= x-1;
zapis skrócony, np.: x++, x--.
- operacja na zmiennej:

zapis klasyczny, np.: x = x+4; x= x*6; x= x/9;
zapis skrócony, np.: x+=4; x*=6; x/=9;

Zwiększanie lub zmniejszanie wartości zmiennej o 1 możliwe jest na dwa sposoby:
a.) zwiększanie/zmniejszanie przed operacją (najpierw zmniejsz/zwiększ wartość zmiennej, później wykonaj operację na tej zmiennej), wówczas notacja operacji jest następująca np.: --x; ++x;
b.) zwiększanie/zmniejszanie po operacji (najpierw wykonaj operację na tej zmiennej a później zmniejsz/zwiększ wartość zmiennej), wówczas notacja operacji jest następująca np.: x--; x++;

Przykład 2.4:

```
//Senat.java

public class Senat{

  public static void main(String args[]){
        int x = 35;
        int s = x++; //warto zmienić kod na ++x i zobaczyć jakie będą wydruki
        System.out.println("Senat Republiki bez planety Naboo składa się z "+ s +" światów");
        System.out.println("Senat Republiki wraz z planetą Naboo składa się z "+ x +" światów");
        x/=6;
        System.out.println("Wszystkie planety senatu mieszczą się w " + x + " galaktykach");
        }

}// koniec public class Senat
```

W Javie nie istnieje możliwość przeciążania operatorów (tzn. dodawania nowego zakresu ich działania). Określono jedynie rozszerzenie operacji dodawania na obiekty typu String. Wówczas możliwe jest wykonanie operacji:

Przykład 2.5:

```
//Relacje.java

public class Relacje{

        public static void main(String args[]){
                String luke = "Luke'a";
                String anakin = "Anakin";
                String relacja = " jest ojcem ";
                String str = anakin+relacja+luke;
                System.out.println(str);
        }

}// koniec public class Relacje
```

generującej następujący komunikat:

Anakin jest ojcem Luke’a.

Operatory przesunięcia działają na bitach w ich reprezentacji poprzez całkowite typy podstawowe danych. Operator „<<„ powoduje przesunięcie w lewo o zadaną liczbę bitów, natomiast operator „>>„ powoduje przesunięcie w prawo o zadaną liczbę bitów, np.:

```
int liczba = 20;
int liczbaL =  liczba << 2;
int liczbaR = liczba>>2;
```

Operatory relacji generują określony rezultat reprezentowany przez typ logiczny *boolean* w wyniku przeprowadzenia porównania:

‘a > b‘ - a większe od b,
‘a < b’ - a mniejsze od b,
‘a >= b‘ - a większe równe jak b,
‘a < =b’ - a mniejsze równe jak b,
‘a == b‘ - a identyczne z b,
‘a != b’ - a różne od b.

Operatory logiczne również generują rezultat reprezentowany przez typ logiczny *boolean*. Rezultat ten jest tworzony w wyniku działania operacji:

‘a && b’ - a i b (rezultatem jest *true* jeśli a i b są *true*);
‘a || b ‘ - a lub b (rezultatem jest *true* jeśli a lub b są *true*).
Operatory bitowe działają podobnie lecz operują na bitach. Ich zapis jest następujący:
‘&’ - bitowy AND,
‘|’ - bitowy OR,
‘^’ - bitowy XOR.

Operator warunkowy w Javie jest skróconą wersją instrukcji warunkowej *if*. Operator ten ma postać:

wyrażenie_boolean ? wartość1 : wartość2;

gdzie:
wyrażenie_boolean oznacza operację generującą jako wynik *true* lub *false*;
wartość1 oznacza działanie podjęte wówczas, gdy wynik wyrażenia jest *true*;
wartość2 oznacza działanie podjęte wówczas, gdy wynik wyrażenia jest *false*.

Przykład 2.6:

```
//Relacje1.java

public class Relacje1{

        public static void main(String args[]){

                System.out.println("Kto to Vader?")
                String vader = "Vader";
                String anakin = "Anakin";
                boolean test =  anakin.equals(vader) ;
                 String s = (vader.equals(anakin)) ?  vader  :  anakin;
                System.out.println(s);
        }

}// koniec public class Relacje1
```

Operacje przypisania polegają na podstawieniu zmiennej (lewa strona - left value - Lvalue) danej wartości, lub wartości innej zmiennej albo wartości wynikowej operacji (prawa strona - right value - Rvalue), np.: a = 12, a = b; a= a+b;.
Dodatkowo należy podkreślić zjawisko tzw. aliasingu, polegające na przypisaniu tego samego uchwytu do dwóch zmiennych, czyli obie wskazują na ten sam obiekt, np.:

Przykład 2.7:

```
//Liczby.java

class Liczba{
        int i;
}

public class Liczby{

        public static void main(String args[]){


                Liczba k = new Liczba();
                Liczba l = new Liczba();
                k.i=4;
                l.i=10;

                System.out.println(" Oto liczby: "+ k.i+", "+l.i);
                k=l;
                System.out.println(" Oto liczby: "+ k.i+", "+l.i);
                l.i=20;
                System.out.println(" Oto liczby: "+ k.i+", "+l.i);



        }

}// koniec public class Liczby
```

2.1.6 Instrukcje sterujące

Instrukcje sterujące służą do sterowanie przepływem wykonywania instrukcji. Do instrukcji sterujących można zaliczyć:

- pętle umożliwiające iteracyjne wykonywanie kodu tak długo aż spełniony jest warunek,
- instrukcje wyboru umożliwiające wybór kodu w zależności od podanego argumentu,
- instrukcje warunkowe umożliwiające wykonanie kodu w zależności od spełnienia podanych warunków,
- instrukcje powrotu umożliwiające przedwczesne zakończenie pętli lub bloku kodu.

Pętla while:

```
while(wyrażenie_logicze)
  wyrażenie
```

Pętla ta powoduje wykonywanie wyrażenia tak długo dopóki rezultat wyrażenia logicznego jest *true*. Wyrażenie stanowi zazwyczaj blok programu wyróżnialny klamrami.

Pętla do-while:

```
do
  wyrażenie
while(wyrażenie_logicze);
```

Podobnie jak pętla *while* sprawdzany jest tu rezultat wyrażenia logicznego. Jeżeli rezultat ten jest *false* wówczas przerywana jest pętla. Różnica pomiędzy *while* i *do-while* polega na tym, że w tej pierwszej warunek pętli sprawdzany jest przed wykonaniem wyrażenia po raz pierwszy.

Petla for:

```
for(inicjowanie; wyrażenie_logicze ; krok)
  wyrażenie
```

Pętla ta powoduje wykonanie wyrażenia tyle razy ile to wynika z warunku przedstawionego w wywołaniu pętli. Warunek ten polega na określeniu wartości startowej iteracji, określeniu końca iteracji oraz kroku. Przykładowo:

```
for (int i =0; i<10; i++){
        System.out.println(„Niech moc będzie z Wami”);
}
```

Dla wszystkich pętli stosować można polecenia *break* i *continue* umieszczane w ciele pętli. Polecenie *break* przerywa pętlę i ją kończy (następuje przejście do kolejnej instrukcji w kodzie). Polecenie *continue* przerywa pętle dla danej iteracji i rozpoczyna następną iterację.

Instrukcja wyboru switch:

```
switch(wyrażenie_wyboru) {
  case wartość1 : wyrażenie; break;
  case wartość2 : wyrażenie; break;
  case wartość3 : wyrażenie; break;
  case wartość4 : wyrażenie; break;
          // …
  default: wyrażenie;
}
```

Instrukcja wyboru *switch* powoduje sprawdzenie stanu wyrażenia_wyboru (zmiennej liczbowej) i w zależności od jej stanu (wartości) wykonywane jest wyrażenie. Słowo *break* oznacza przerwanie działania w instrukcji wyboru (nie jest wykonywane kolejne wyrażenie). Domyślne wyrażenie jest wykonywane dla wszystkich innych stanów niż te wymienione w ciele instrukcji.

Instrukcja warunkowa if:

```
if (wyrażenie_logiczne))  { wyrażenie } else { wyrażenie};
```

Instrukcja warunkowa *if* sprawdza stan logiczny wyrażenia logicznego i jeżeli jest *true* to wykonywane jest pierwsze wyrażenie, jeśli *false* to wykonywane jest wyrażenie po słowie kluczowym *else*.

Instrukcja powrotu *return*:

```
return (wyrażenie);
```

Instrukcja powrotu *return* kończy metodę, w ciele której się znajduje i powoduje przeniesienie wartości wyrażenia do kodu wywołującego daną metodę. Oznacza to, że typ wartości wyrażenia instrukcji *return* musi być zgodny z typem zadeklarowanym w czasie definicji metody, w ciele której znajduje się instrukcja *return*.

Prześledźmy dwa przykłady obrazujące działanie pętli i instrukcji warunkowych.

Przykład 2.8:

```
//PetleCzasu.java


public class PetleCzasu{

        public static void main(String args[]){

                int i=0;
                do{
                        i++;
                        if (i==10) break;
                        System.out.println("Moc jest z Wami");
                }while(true);

                for (int j=0; j<10; j++){
                        if ((j==2) || (j==4)) continue;
                        System.out.println("Moc jest ze MNA ");
                }

        }

}// koniec public class PetleCzasu
```

Powyższy program demonstruje działanie pętli *do-while*, pętli for oraz instrukcji warunkowej *if*. W czasie obsługi pierwszej pętli ustawiono warunek logiczny na *true*. Oznacza to, że pętla będzie wykonywała się w nieskończoność o ile w jej ciele nie nastąpi przerwanie pętli. Przerwanie pętli w przykładzie 2.8 jest wykonane poprzez wywołanie instrukcji *break*, po osiągnięciu przez wskaźnik iteracji wartości 10. Przerwanie następuje przed poleceniem wydruku komunikat, tak więc zaledwie 9 razy zostanie wyświetlona wiadomość: **Moc jest z Wami**. Druga pętla przykładu przebiega przez 10 iteracji, niemniej dla iteracji nr 2 i 4 generowane jest polecenie *continue*, które powoduje przerwanie pętli dla danej iteracji i rozpoczęcie nowej. Dlatego komunikat obsługiwany w tej pętli: **Moc jest ze MNA**, zostanie wyświetlony zaledwie 8 razy. Kolejny przykład ukazuje zasadę działania instrukcji wyboru *switch*.

Przykład 2.9:

```
//Wybor.java

public class Wybor{

	public static void main(String args[]) throws Exception{

		System.out.println("Wybierz liczbę mieczy: "+
					"1, 2, 3 lub 4 i naciśnij Enter");
		int test = System.in.read();
		switch(test){
			case 49:
					System.out.println("Wybrano 1");
					break;
			case 50:
					System.out.println("Wybrano 2");
					break;
			case 51:
					System.out.println("Wybrano 3");
					break;
			case 52:
					System.out.println("Wybrano 4");
					/* Brak break; program przejdzie do
					pola default i wykona odpowienie
					instrukcje */
			default:
					System.out.println("Wybrano cos");
		}//koniec switch

	}

}// koniec public class Wybor
```

Powyższy przykład ukazuje działanie instrukcji wyboru *switch*. Na początku przykładowego programu pobierana jest wartość liczby całkowitej będącej reprezentacją znaku przycisku klawiatury wciśniętego przez użytkownika programu. Pobranie wartości znaku jest wykonane poprzez otwarcie standardowego strumienia wejściowego (*System.in* - > otwarty *InputStrem*) i wywołanie metody *read*(), zwracającej kod znaku jako liczbę *int* w przedziale 0-255. Następnie w zależności od wartości pobranej liczby generowana jest odpowiednia wiadomość na ekranie monitora. Po obsłudze każdego wyrażenia wyboru z wyjątkiem liczby 52 (kod znaku '1') umieszczana jest instrukcja przerwania *break*. Brak tej instrukcji powoduje uruchomienie wyrażeń znajdujących się w obsłudze kolejnego wyrażenia wyboru (np.: obsługa wartości 52). W przypadku, kiedy wciśnięty klawisz klawiatury różni się od 1, 2, 3 czy 4 obsłużony zostanie standardowy warunek wyboru *default*.

2.2 Informacja praktyczna – wyświetlanie polskich znaków w konsoli platformy Javy

Jak można było zauważyć w pracy z poprzednimi programami przykładowymi wszelkie komunikaty zawierające polskie litery wyświetlane były z „krzakami" zamiast polskich liter. Warto rozważyć ten problem, ponieważ szereg aplikacji może wymagać używania polskich liter.

Po pierwsze istotne jest uświadomienie sobie w jakim systemie kodowania znaków stworzony został kod źródłowy programu. Jest to istotne, ponieważ często stosuje się w kodzie źródłowym specyficzne znaki języka polskiego. Wiele programów zawiera w swym kodzie źródłowym zainicjowane zmienne typu *char* lub *String* zawierające takie znaki, np.:

```
(...)
String str = „W mroku jawił się tylko żółty odcień świetlistego miecza”;
(...)
```

Tworząc kod źródłowy zawierający specyficzne znaki języka należy zwrócić uwagę na to, jaki edytor tekstu albo w jakim systemie kodowania znaków zapisany został kod źródłowy. Dla MS Windows PL standardową metodą kodowania znaków jest system „Cp1250”, często reprezentowany w edytorze jako kod ANSI. Natomiast dla środowiska MS DOS PL wykorzystywaną stroną kodową jest Cp852. Jeszcze inaczej może być w środowisku UNIX gdzie wykorzystuje się „polską” stronę kodową ISO-8859-2. Zapisując kod źródłowy dokonujemy konwersji znaków na bajty zgodnie z wybraną (lub standardową dla danej platformy) stroną kodową. Zestaw bajtów jest zapisywany w pliku i może być następnie wykorzystywany do wyświetlenia jako tekst po konwersji bajtów na znaki (wedle wybranej strony kodowej). Przykładowo tworząc tekst w środowisku DOS PL np. korzystając z programu edit, zawierający polskie znaki, następnie nagrywając plik z tym tekstem otrzymamy zbiór bajtów odpowiadający znakom według strony kodowej Cp852. Otwarcie tego pliku w środowisku Windows PL wybierając standardową stronę kodową (Cp1250) spowoduje wyświetlenie tekstu bez polskich liter. Jest to oczywiście spowodowane tym, że zapisane w pliku bajty kodowane przy wyświetlaniu według innego kodu znaków niż przy tworzeniu będą reprezentowane przez inne znaki. Podobna sytuacja wystąpi przy każdej innej zamianie stron kodowych. Ponieważ Java używa systemu kodowania znaków Unicode do przechowywania wszelkich zmiennych typu char lub String, dlatego w czasie kompilacji kodu wszystko jest zamieniane na Unicode. Oznacza to, że również zmienne typu char i String są zamieniane na Unicode. Konwersja ta przebiega następująco:

a.) kompilator czyta bajty zapisane w pliku kodu źródłowego,
b.) kompilator sprawdza zmienną środowiska („file.encoding”) oznaczającą stronę kodową dla danej platformy (System.getProperty(„file.encoding”);)
c.) kompilator dokonuje konwersji bajtów na znaki według strony kodowej danje platformy,
d.) kompilator dokonuje konwersji powstałych znaków na odpowiadające im kody znaków w Unicodzie (16-bitowe).

Ponieważ kompilator dokonuje konwersji bajtów z pliku poprzez stronę kodową platformy na kody w Unicodzie istotna jest zgodność kodowania znaków przy tworzeniu pliku (z kodem źródłowym) z kodowaniem przy konwersji znaków na Unicode. Standardową stronę kodową platformy można uzyskać jako własność systemu (zmienna Maszyny Wirtualnej) poprzez wywołanie metody System.getProperty(„file.encoding”). Dla platformy Windows PL jest to oczywiście „Cp1250”. Programista, pisząc kod źródłowy zawierający specyficzne znaki języka polskiego po czym nagrywając go do pliku zgodnie ze stroną kodową Cp1250 (ANSI) otrzymuje gotowy do kompilacji w standardowym środowisku Javy (Windows PL) zbiór bajtów. Jeżeli plik źródłowy został zapisany zgodnie z inną stroną kodową niż standardowa strona kodowa platformy (np. plik nagrano w środowisku DOS PL - CP852) to wywołanie kompilatora musi jawnie wskazywać na sposób kodowania użyty do stworzenia zbioru bajtów kodu źródłowego, np.:

Javac –g –encoding „Cp852” NazwaProgramu.java

Ponieważ specyficzne znaki języka polskiego są jeszcze inaczej kodowane na maszynach UNIX (ISO 8859-2), to właściwe wydaje się używanie Unicodu do zapisu polskich liter w źródle programu. Można to robić bezpośrednio wpisując znaki ucieczki, np. /u0105 zamiast znaku ‘ą’, lub poprzez wprowadzenie automatycznej konwersji plików wykorzystując jednolity system kodowania. Kody specyficznych znaków języka polskiego w Unicodzie są następujące:

Znak	Kod heksadecymalny	Kod dziesiętny
ą	0105	261
ć	0107	263
ę	0119	281
ł	0142	322
ń	0144	324
ó	00F3	243
ś	015B	339
ź	017A	378
ż	017C	380
Ą	0104	260
Ć	0106	262
Ę	0118	280
Ł	0141	321
Ń	0143	323
Ó	00D3	211
Ś	015A	346
Ź	0179	377
Ż	017B	379

Stworzenie odpowiednich kodów Unicodu w skompilowanych plikach B-kodu nie stanowi jeszcze rozwiązania problemu wyświetlania polskich znaków w konsoli Javy. Otóż Maszyna Wirtualna wykonując kod zapisany w pliku dokonuje konwersji zmiennych znakowych z Unicodu na kod właściwy dla danej platformy. Dla Maszyny Wirtualnej pracującej w środowisku Windows PL standardową stroną kodową jest „Cp1250”. Oznacza to, że znaki zapisane kodowo w Unicodzie podlegają konwersji na bajty dla strony kodowej „Cp1250”. W przypadku wyświetlania wiadomości na ekranie bajty te są wysyłane do standardowego strumienia wyjścia (do konsoli) gdzie są interpretowane i wyświetlane jako znaki. W tym momencie występuje pewien problem. Otóż dla danych ustawień lokalnych (Strefa Czasowa i inne ustawienia lokalne) dla Polski stroną kodową platformy jest „Cp1250” lecz stroną kodową konsoli (konsola DOS-u) jest „Cp852”. Dla tak zdefiniowanej konsoli bajty znaków powstałe poprzez kodowanie w „Cp1250” nie spowodują wyświetlenia polskich znaków lecz otrzymamy inne znaki językowe. Jakie jest więc rozwiązanie tego problemu? Otóż trzeba stworzyć własny strumień wyjścia obsługujący żądaną stronę kodową. W rozważanym przypadku będzie to oczywiście „Cp852”. Do konstrukcji nowego strumienia wyjścia można użyć klasy *OutputStreamWriter,* której konstruktor umożliwia podanie metody kodowania znaków np.: *PrintWriter o = new PrintWriter(*

new OutputStreamWriter(System.out,"Cp852"), true);. Następujący przykład ukazuje sposób wykorzystania tej klasy.

Przykład 2.10:

```
//ZnakiPL.java

import java.io.*;

public class ZnakiPL{

 public static void main(String args[]){

  String st = "ślążóżźćń";
  char zn = 'ą';
  byte c[] = new byte[10];
  byte d[] = new byte[10];
  String st1="nic";
  String st2="nic";
  String st3="nic";
  String st4="nic";


  try{

            PrintWriter o = new PrintWriter( new OutputStreamWriter(System.out,"Cp852"), true);
            System.out.println("Kodowanie to: "+System.getProperty("file.encoding"));


            c=st.getBytes("Cp852");
            d=st.getBytes("Cp1250");
            o.println("W Unicodzie \u0105 to: "+(int)zn);
            o.println("Je\u015Bli prezentowana liczba jest r\u00F3\u017Cna od 261 to oznacza,");
            o.println("\u017Ce kod programu napisano z inn\u0105 stron\u0105 kodow\u0105 ni\u017C ta,");
            o.println("kt\u00F3r\u0105 u\u017Cyto w kompilacji (standardowo dla Windows PL->Cp1250");

            st2 = new String(c, "Cp852");
            st1 = new String(d,"Cp1250");
            st3 = new String(c,"Cp1250");
            st4 = new String(d,"Cp852");

            o.println("Przejście: Cp852(bajty)->Cp852(String)->Unicode(Java)->Cp852(strumien)daje: " + st2);
            o.println("Przejście: Cp1250(bajty)->Cp1250(String)->Unicode(Java)->Cp852(strumien)daje: " +st1);
            o.println("Przejście: Cp852(bajty)->Cp1250(String)->Unicode(Java)->Cp852(strumien)daje: " +st3);
            o.println("Przejście: Cp1250(bajty)->Cp852(String)->Unicode(Java)->Cp852(strumien)daje: " +st4);

            o.println("Przejście: Cp1250(String)->Unicode(Java)->Cp852(strumien)daje: " +st);
            //wyświetla polskie znaki w konsoli DOS
            System.out.println(" ");
            System.out.println("Przejście: Cp852(bajty)->Cp852(String)->Unicode(Java)->Cp1250(strumien)daje: " + st2);
            System.out.println("Przejście: Cp1250(bajty)->Cp1250(String)->Unicode(Java)->Cp1250(strumien)daje: " +st1);
            System.out.println("Przejście: Cp852(bajty)->Cp1250(String)->Unicode(Java)->Cp1250(strumien)daje: " +st3);
            System.out.println("Przejście: Cp1250(bajty)->Cp852(String)->Unicode(Java)->Cp1250(strumien)daje: " +st4);

            System.out.println("Przejście: Cp1250(String)->Unicode(Java)->Cp1250(strumien)daje: " +st);

            System.out.println(" ");
            String st5=          "Oto Unicode: "+ "\u0104\u0105\u0142\u0144\u00F3\u015B ";
            System.out.println("Przejście: Unicode(String)->Unicode(Java)->Cp1250(strumien)daje: " + st5);
            o.println("Przejście: Unicode(String)->Unicode(Java)->Cp852(strumien)daje: " + st5);
```

```
 } catch (Exception e){
          e.printStackTrace();
 }

}// koniec main()

}// koniec public class ZnakiPL
```

Podobne problemy i sposób rozwiązania można wykorzystać przy innych przejściach pomiędzy stronami kodowymi, oraz dla obsługi innych strumieni np.: plików.