Język JAVA

podstawy programowania

[43] Na ogół łatwiej daje się człowiek przekonać racjom, do których sam doszedł, niż tym, które nastręczyły się komuś innemu.

„Myśli” Blaise Pascal

© Jacek Rumiński, 1999

Wprowadzenie

Język JAVA jest niewątpliwie najbardziej rozwijającym się obecnie środowiskiem tworzenia aplikacji. Czerpie on to co najlepsze z takich języków jak C++ czy Smaltalk przy zdecydowanie prostszej i bardziej czytelnej składni (konstrukcji) programów. Zawiera elementy programowania zarówno strukturalnego jak i obiektowego, zdarzeniowego jak i współbieżnego. Poprzez standardowe jak i rozszerzone biblioteki wkracza w różnorodne rejony zastosowań takie jak np. karty inteligentne i elektronika, systemy zarządzania bazami danych, obsługa multimediów, Internet, grafika 3D, kryptografia, itd. Co więcej JAVA jest niespotykanie bezpiecznym środowiskiem i umożliwia w znaczny sposób kontrolę i sterowanie bezpieczeństwem. Zdecydowanie różni się od innych języków trzeciej generacji tym, że jest językiem interpretowanym a nie kompilowanym. Oznacza to, że powstały w wyniku kompilacji kod wynikowy nie jest programem jaki można niezależnie uruchomić lecz stanowi tzw. Beta-kod, który jest interpretowany przez Maszynę Wirtualną (JavaVM) pracującą w określonym środowisku. Ze względu na kod nie istotne jest na jakim sprzęcie będzie uruchamiana aplikacja. Ważna jest tylko Maszyna Wirtualna. Jest to niezwykle ciekawy pomysł umożliwiający odcięcie się od wszystkich poziomów sprzętowo-programowych będących poniżej Maszyny Wirtualnej. Koncepcja ta jest powszechna również w samym języku JAVA, dzięki czemu poprzez stworzenie abstrakcyjnych klas i metod podstawowe biblioteki Javy nie muszą być nieustannie rozbudowywane.
JAVA jest niewątpliwie językiem najbliższej przyszłości, warto więc poświęcić mu trochę czasu.

Krótka historia Javy

1990 - Bill Joy w raporcie „Further” sugeruje SUNowi 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 Javą 1.0, Sun propagugę darmowe środowisko JDK 1.0,

1999 - Java 2 Nowe oblicze Javy.

Główne źródła informacji o Javie w Internecie

http://java.sun.com, SUN, The source of Java Technology (źródło technologii Javy), a w szczególności:

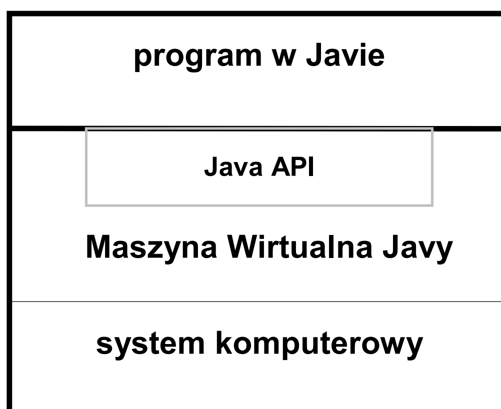
http://java.sun.com/docs/books/jls/, Gosling J., Joy B, Steele G. The Java Language Specification. Addison-Wesley, 1996, (specyfikacja języka !!!),
http://java.sun.com/products/jdk/1.2, SUN, Java 2 SDK software and documentation site (strona źródłowa oprogramowania i dokumentacji Javy),
http://developer.java.sun.com/developer/infodocs/, SUN, On-line books & Tutorials (książki i podręczniki do nauki Javy),
http://java.sun.com/docs/books/tutorial, SUN, The Java Tutorial (Podręcznik Javy).

Forum użytkowników Javy, poradniki, testy, doświadczenia:
http://www.javaworld.com,
http://www.javareport.com,
http://www.jars.com,
http://www.gamelon.com,
http://www.javalobby.com

i wiele innych, których źródła nie można nawet wymienić.

Java platformą tworzenia i wykonywania aplikacji

Platformą nazywa się przeważnie pewną kombinację sprzętu i oprogramowania umożliwiającą tworzenie i wykonywanie programów. Przyjęło się powszechnie mówić o tzw. platformie sprzętowo-programowej. Platformę stanowi więc komputer o danej konfiguracji oraz system operacyjny w środowisku którego uruchamiana jest dowolna aplikacja. Przykładowe platformy to Intel PC + Windows NT; Sun Ultra + Solaris; SGI O2 + Irix 6.4, itp. Konstrukcja platformy Javy jest podobna, niemniej nie odnosi się bezpośrednio do sprzętu i systemu operacyjnego, które stanowią dla niej pewną abstrakcję. Istotą platformy Javy jest zbiór dwóch elementów: Java API (Application Programming Interfaces) - interfejsy tworzenia aplikcaji oraz JavaVM (Virtual Machine) - maszyna wirtualna. Maszyna wirtualna Javy jest rozwinięciem dotychczas używanego pojęcia platformy, stanowiąc pewną nadbudowę. Maszyna wirtualna interpretuje kod wynikowy (Beta-kod) Javy do kodu wykonywalnego danego systemu operacyjnego i komputera, którego jest nadbudową. Oznacza to, że Maszyna Wirtualna jest interfejsem pomiędzy uniwersalnym kodem Javy, a różnymi konfiguracjami komputerów. Ta różnorodność systemów komputerowych wymaga różnorodności Maszyn Wirtualnych. Firma Sun dostarcza obecnie Maszynę Wirtualną wersji Java 2 dla systemów operacyjnych Windows95/98/NT oraz Solaris. Oczywiście interpretacja kodu właściwa dla danego systemu operacyjnego (konwersja w locie Beta-kodu do kodu wykonywalnego) wymaga odpowiednich bibliotek. Biblioteki klas, metod, pól, itp., zarówno te zależne sprzętowo jak i te niezależne sprzętowo stworzone już w Javie znajdują się w postaci skompilowanej w Java API. Podsumowując Java VM oraz Java API tworzą platformę Javy zwane często środowiskiem uruchomieniowym aplikacji - Java Runtime Engine (JRE).



Rysunek 1. Środowisko uruchomieniowe programu w Javie - platforma Javy.

Java - środowisko tworzenia aplikacji

Aby można było uruchomić aplikację na platformie Javy trzeba ją najpierw stworzyć, po czym skompilować do Beta-kodu. Posługując się regułami języka Java oraz zapleczem klas i metod powstaje kod źródłowy programu. W celu generacji Beta-kodu program ten podaje się następnie kompilacji (np. kompilatorem „java" firmy Sun w pełni stworzonego za pomocą języka Java). Dla potrzeb tworzenia aplikacji SUN oferuje pakiet Java Development Kit (JDK lub Java SDK - Software Development Kit), który składa się z JRE oraz narzędzi kompilacji i bibliotek.

Java - język programowania

Stworzenie programu w Javie polega na umiejętnym wykorzystaniu znajomości reguł języka oraz bibliotek. Konstrukcja programu i reguły języka przypominają znacznie język C. Programiści znający ten język z łatwością i przyjemnością rozpoczną pracę z Javą. Dla znawców języka C miła będzie informacja, że w Javie nie używa się w ogóle wskaźników, statycznego rezerwowania i zwalniania pamięci itp. Program w Javie nie zawiesi się więc z uwagi na „Null Pointer Assigment". Bardzo istotne w konstrukcji programu jest znajomość obsługiwanych typów. Ponieważ kod Javy jest niezależny od sprzętu, to również typy danych są niezależne od sprzętu (platformy). Java jest językiem obiektowym, dzięki czemu kod jest uniwersalny i bardzo czytelny. Nowością jaką niesie ze sobą Java jest również tworzenie tzw. apletów. Aplet jest programem wykonywanym w określonych ramach (nadbudowie Maszyny Wirtualnej). Aplet ma więc takie możliwości jakie nadaje mu program uruchomieniowy. Przykładowe programy uruchomieniowe to przeglądarki WWW np. Netscape, Internet Explorer. Kolejnym nowym elementem konstrukcyjnym Javy jest to, że można kod grupować w liczne wątki, które w wyniku interpretacji tworzą niezależnie wykonywane procesy współdzielące czas procesora.

Opis środowiska Java 2 SDK

Proces tworzenia aplikacji Javy z pomocą dostarczanego przez Suna środowiska Java 2 SDK można przedstawić następująco:

1. Napisanie z pomocą dowolnego edytora tekstu kodu źródłowego programu zawierającego klasę publiczną o nazwie takiej samej (dokładnie takiej samej z uwzględnieniem wielkości znaków) jak docelowa nazwa programu np. RycerzJedi.

2. Nagranie kodu źródłowego jako pliku o danej nazwie z rozszerzeniem .java, np. RycerzJedi.java

3. Kompilacja kodu źródłowego zawartego w pliku z rozszerzeniem .java do pliku docelowego o rozszerzeniu .class zawierającego Beta-kod np.
c:\ javac RycerzJedi.java
gdzie:
javac - nazwa komilatora programów Javy stworzonego przez Suna (kompilator napisany w Javie),
RycerzJedi.java - kod źródłowy programu do kompilacji (WAŻNE: podana nazwa pliku musi zawierać rozszerzenie .java).
W Wyniku kompilacji powstanie plik lub zestaw plików z tym samym trzonem nazwy o rozszerzeniu .class, np. RycerzJedi.class.

4. Uruchomienie w środowisku interpretatora Beta-kodu, np.
c:\ java RycerzJedi
gdzie:
java - nazwa interpretatora Javy stworzonego przez Suna, inaczej uruchomienie Maszyny Wirtualnej,
RycerzJedi - nazwa pliku z Beta-kodem programu w Javie kompilacji (WAŻNE: podana nazwa pliku nie może zawierać rozszerzenia .class).

W celu kompilacji i uruchomienia programu napisanego w języku Java użyto w powyższym przykładzie dwóch podstawowych narzędzi pakietu Java 2 SDK: javac oraz java. Kompilator „javac” (często nazywany „Jawak”) jest nieodzowną częścią pakietu SDK, podczas gdy interpretator „java” stanowi specyficzną dl adanej platformy część pakietu środowiska uruchomieniowego Java Runtime Engine. Wynika stąd, że po instalacji pakietu Java SDK interpretator „java” będzie znajdował się zarówno w części JRE (niezależnej od tworzenia aplikacji) jak i w zbiorze narzędzi tworzenia aplikacji. Przykładowo katalog zawierający Java 2 SDK wygląda następująco:

<DIR> bin
<DIR> demo
<DIR> include
<DIR> include-old
<DIR> jre
<DIR> lib
935 COPYRIGHT

8˙762 LICENSE
6˙010 README
9˙431 README.html
16˙715˙279 src.jar
313˙746 Uninst.isu

W katalogu „bin” znajdują się liczne narzędzia obsługi aplikacji np:

javac - kompilator,
java - interpretator z konsolą ,
javaw - interpretator bez konsoli,
javadoc - generator dokumentacji API,
appletviewer - interpretator apletów,
jar - zarządzanie plikami archiwów (JAR),
jdb - debager,

Ze względu na to, że pisząc programy w Javie często korzysta się z narzędzi znajdujących się w katalogu „bin”, warto ustawić w środowisku ścieżkę dostępu do tego katalogu. Narzędzia dostępne w tym katalogu można wywoływać z licznymi opcjami. Praktycznie jednak najbardziej przydatne opcje to:

***javac:**
-g ->wyświetl pełną informację debagera,
- verbose ->wyświetl wszystkie komunikaty w czasie kompilacji,
np. javac -g -verbose RycerzJedi.java
***java:**
-cp -classpath -> gdzie -classpath katalog zawierający wykorzystywane klasy użytkownika (lepiej ustawić zmienną środowiska CLASSPATH),
-version ->wyświetl wersję platformy Javy.

Drugim ważnym katalogiem jest katalog „jre”. Jak łatwo się domyślić w katalogu tym znajduje się Java Runtime Environment JRE - platforma Javy. Zgodnie z tym co powiedziano na początku platforma Javy składa się z Maszyny Wirtualnej oraz bibliotek API. Dlatego katalog „jre” podzielony jest na dwa podkatalogi: „bin” - w którym znajduje się interpretator „java” (ten sam co wcześniej) oraz : „lib” gdzie znajdują się spakowane biblioteki API oraz pliki konfiguracyjne i środowiskowe platformy (np. określające poziom bezpieczeństwa, czcionki, fonty, itp.).

Uwaga praktyczna:

W czasie nauki języka Java lepiej unikać wszelkiego rodzaju programów typu szybkiego tworzenia aplikacji, gdyż traci się często kontrolę nad zrozumieniem treści tworzonego programu.

Integralną częścią środowiska Javy są biblioteki. Korzystanie z bibliotek jest znacznie prostsze jeśli rozumie się jak z nich korzystać. Nieodzownym jest więc korzystanie z dokumentacji bibliotek API. Opis bibliotek jest dostępny oddzielnie względem środowiska JDK i stworzony jest jako serwis WWW, który można przeglądać on-line lub off-line. Dokumentacja zawiera spis wszystkich pakietów, klas, ich pól i metod oraz wyjątków wraz z odpowiednimi opisami.

Rozdział 1 Tworzenie i obsługa programów

1.1 Klasy obiektów
1.2 Aplikacje
1.3 Aplety
1.4 Aplikacja i aplet w jednym kodzie
1.5 Interpretacja, kompilacja i obsługa klas w Javie
1.6 Wyjątki
1.7 Klasy wewnętrzne

1.1 Klasy obiektów

Ponieważ Java jest językiem obiektowym operuje na pojęciach ogólnych. Znane z historii filozofii pojęcie uniwersalii, a więc pewnych klas ogólnych bytów np. człowiek, małpa, rycerz Jedi, jest stosowane w Javie (jak i w innych językach obiektowych). Klasa bytów (class) określa opis bytu (stałe, zmienne, pola) oraz jego zachowanie (metody). Przykładowo klasę ludzi można opisać poprzez parametry: kolor oczu, rasę, kolor włosów, itp., (pola klasy) ; oraz poprzez zachowanie: chodzi, oddycha, itp, (metody klasy). Obiektem jest konkretna realizacja klasy, a więc np. konkretny człowiek - Kowalski, a nie sama klasa. Można upraszczając podać generalny podział na rodzaj kodu strukturalny i obiektowy. W ujęciu strukturalnym w celu określenia zbioru konkretnych osób należy zdefiniować zbiór (wektor, macierz) klas (struktur), gdzie każda klasa opisuje jedną osobę. W podejściu obiektowym w celu określenia zbioru konkretnych osób należy zainicjować zbiór obiektów tej samej klasy. Odwołanie się do parametru klasy w podejściu strukturalnym odbędzie się poprzez odniesie przez nazwę odpowiedniej klasy, natomiast w podejściu obiektowym poprzez nazwę obiektu. Oznacza to, że w aplikacjach obiektowych bez inicjacji obiektu nie można odwołać się do parametrów (pól) i metod klasy. Wyjątkiem są te pola i metody, które oznaczone są jako statyczne (static). Pole statyczne, bez względu na liczbę obiektów danej klasy, będzie miało tylko jedną wartość. Pola i metody statyczne mogą być wywoływane przez referencję do klasy. Po tym podstawowym wprowadzeniu zapoznajmy się z zasadniczą konstrukcją programu w Javie (rozważania dotyczące Javy jako Języka obiektowego znajdują się w dalszej części tego dokumentu).

Klasę w Javie konstruuje się zaczynając od stworzenia ramy klasy:

```
class NazwaKlasy {
	// pola
	typ_zmiennej zmienna;
	.
	.
	.
	typ_zmiennej zmienna;

	//konstruktor - metoda o tej samej nazwie co klasa - wywoływana
	//automatycznie przy tworzeniu obiektu danej klasy
	NazwaKlasy(typ_argumentu nazwa_argumentu){
		treść_konstruktora;
	}
```

```
    //metody
    typ_wartości_zwracanej nazwa_metody(typ_argumentu nazwa_argumentu){
        treść_metody;
    }
}// koniec class NazwaKlasy
```

Przykład 1.1:

```
//RycerzJedi.java

//klasa
class RycerzJedi{

    //pola
    String nazwa;
    String kolor_miecza;

    //konstruktor
    RycerzJedi(String nazwa, String kolor_miecza){
        this.nazwa=nazwa;
        this.kolor_miecza=kolor_miecza;
    }

    //metody
    void opis(){
        System.out.println(”Rycerz ”+nazwa+ ” ma ”+kolor_miecza+” miecz.”);
    }

}// koniec class RycerzJedi
```

Warto zwrócić uwagę na zastosowaną już metodę notacji. Otóż przyjęło się w Javie (specyfikacja języka Java) oznaczanie nazwy klasy poprzez stosowanie wielkich liter dla pierwszej litery i tych, które rozpoczynają nowe znaczenie w nazwie klasy. Metody i obiekty oznacza się z małych liter. Warto również stosować prosty komentarz rozpoczynający się od znaków „//” i mający zasięg jednego wiersza. Dalej w tekście zostaną omówione inne, ważne zagadnienia związane z konstrukcją i opisem kodu.

Przedstawiona powyżej rama i przykład klasy nie mogą być wprost uruchomione. Konieczne jest utworzenie elementu wywołującego określone działanie. W Javie jest to możliwe na dwa sposoby. Po pierwsze za pomocą aplikacji, po drugie appletu.

1.2 Aplikacje

Aplikacja w języku Java to zdefiniowana klasa publiczna wraz z jedną ściśle określoną metodą statyczną o formie:

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

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

Tablica „args[]” będąca argumentem metody main() jest zbiorem argumentów wywołania aplikacji w ciele której znajduje się metoda. Kolejność argumentów jest następująca: argument 1 wywołania umieszczany jest w args[0], argument 2 wywołania umieszczany jest w args[1], itd. Występująca nazwa „args” oznacza dynamicznie tworzony obiekt, który zawiera args.length elementów typu łańcuch znaków. Pole „length” oznacz więc liczbę elementów tablicy. Łatwo jest więc określić argumenty oraz ich liczbę korzystając z obiektu „args”. Warto zwrócić uwagę na to, że oznaczenie „length” jest własnością tablicy (ukrytym polem każdej tablicy) a nie metodą obiektu args. Istnieje metoda klasy String o nazwie „length()” (metody zawsze są zakończone nawiasami pustymi lub przechowującymi argumenty), łatwo więc się pomylić. Dla zapamiętania różnicy posłużmy się prostym przykładem wywołania:
args.length - oznacz ilość elementów tablicy „args”;
args[0].length() -oznacza rozmiar zmiennej tekstowej o indeksie 0 w tablicy „args”.

Prosta aplikacja działająca w Javie będzie miała następującą formę:

Przykład 1.2:

```
// Jedi.java :

public class Jedi{
        public static void main(String args[]){
                System.out.println(”Rycerz Luke ma niebieski miecz.”);
        }// koniec public static void main(String args[])
}// koniec public class Jedi
```

Nagrywając powyższy kod do plik „Jedi.java”, kompilując poprzez podanie polecenia: „javac -g -verbose Jedi.java” (pamiętać należy że kompilację należy wykonać z scieżki występowania pliku źródłowego, lub należy mieć odpowiednio ustawioną zmienną środowiska CLASSPATH) można wykonać powstały Beta-kod „Jedi.class” używając interpretatora: „java Jedi”. Uzyskamy następujący efekt:

Rycerz Luke ma niebieski miecz.

Jak działa aplikacja z przykładu 1.2? Otóż w ustalonej ramie klasy, w metodzie programu uruchomiono polecenie wysłania do strumienia wyjścia (na standardowe urządzenie wyjścia - monitor) łańcuch znaków „Rycerz Luke ma niebieski miecz.”. Konstrukcja ta wymaga krótkiego komentarza. Słowo „System” występujące w poleceniu wydruku oznacza statyczne odwołanie się do elementu klasy System. Tym elementem jest pole o nazwie „out”, które stanowi zainicjowany obiekt typu PrintStream (strumień wydruku). Jedną z metod klasy PrintStream jest metoda o nazwie „println”, która wyświetla w formie tekstu podaną wartość argumentu oraz powoduje przejście do nowej linii (wysyłany znak końca linii). W podanym przykładzie nie ukazano jawnie tworzenia obiektu. Można więc powyższy przykład nieco rozwinąć.

Przykład 1.3

```
// Jedi1.java :

//klasa
class RycerzJedi{

	//pola
	String nazwa;
	String kolor_miecza;

	//konstruktor
	RycerzJedi(String nazwa, String kolor_miecza){
		this.nazwa=nazwa;
		this.kolor_miecza=kolor_miecza;
	}

	//metody
	void opis(){
		System.out.println(”Rycerz ”+nazwa+ ” ma ”+kolor_miecza+” miecz.”);
	}

}// koniec class RycerzJedi

public class Jedi1{

	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 Jedi1
```

Zapiszmy kod aplikacji pod nazwą Jedi1.java, a następnie wykonajmy kompilację z użyciem polecenia: „javac -g -verbose Jedi1.java”. Zwróćmy uwagę na wyświetlane w czasie kompilacji komunikaty. Po pierwsze kompilator ładuje wykorzystywane przez nas klasy. Klasy te znajdują się w podkatalogu „lib” katalogu „jre” (biblioteki Java Runtime Engine). Klasy są pogrupowane w pakiety tematyczne np. „lang” czy „io”. Wszystkie grupy klas znajdują się w jednym zbiorze o nazwie „java”. Całość jest spakowana metodą JAR (opracowaną dla Javy) do pliku „rt.jar”. Wszystkie prekomilowane klasy znajdujące się w pliku „rt.jar” stanowią podstawowe biblioteki języka Java. Biblioteki te ujęte w pakiety wywoływane są następująco:
np.:
java.lang.*; - oznacza wszstkie klasy (interfejsy i wyjątki - o czym później) grupy lang, głównej biblioteki języka Java,
java.io.*; - oznacza wszstkie klasy (interfejsy i wyjątki - o czym później) grupy io (wejście/wyjście), głównej biblioteki języka Java,
java.net.Socket; - oznacza klasę Socket grupy net (sieć), głównej biblioteki języka Java.
Wykorzystując różne klasy biblioteki języka Java należy pamiętać w jakim pakiecie się znajdują. Wszystkie klasy poza tymi zawartymi w pakiecie „java.lang.*”

wymagają zastosowania jawnego importu pakietu (klasy) w kodzie programu. Odbywa się to poprzez dodanie na początku kodu źródłowego linii:
np.:

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

Po dodaniu takiej linii kompilator wie gdzie szukać używanych w kodzie klas. Pakiety zaczynające się od słowa „java" oznaczają zasadnicze pakiety języka Java. Niemniej możliwe są różne inne pakiety, których klasy można stosować np.: „org.omg.CORBA". Nazewnictwo pakietów jest określone w specyfikacji języka i opiera się o domeny Internetowe. Oczywiście można tworzyć własne pakiety. Występujące w nazwie pakietu kropki oznaczają zmianę poziomu w drzewie katalogów przechowywania klas. Przykładowo „org.omg.CORBA.Context" oznacza, że w katalogu „org", w podkatalogu „omg", w podkatalogu „CORBA" znajduje się klasa „Context.class"; czyli zapis pakietu jest równoważny w sensie systemu plików: /org/omg/CORBA/Context.class.
Obserwując dalej komunikaty wyprodukowane przez kompilator Javy łatwo zauważyć, że zapisane zostały dwie klasy: RycerzJedi.class oraz Jedi1.class. Oddzielnie definiowane klasy zawsze będą oddzielnie zapisywane w Beta-kodzie.
Po kompilacji należy uruchomić przykład 1.3. W tym celu wywołajmy interpretator: „java Jedi1". Zdarza się czasem, że wykorzystywany Beta-kod klas pomocniczych (np. klasa RycerzJedi.class) znajduje się w innym miejscu dysku. Wówczas otrzymamy w czasie wywołania aplikacji błąd o niezdefiniowanej klasie. Należy wtedy ustawić odpowiednio zmienną środowiska CLASSPATH lub wywołać odpowiednio interpretator:
„java -cp <ścieżka dostępu do klas> NAZWA_KLASY_GŁÓWNEJ".

Prawidłowe działanie aplikacji z przykładu 1.3 da następujący rezultat:

Rycerz Luke ma niebieski miecz.
Rycerz Obi-wan ma biały miecz.

Prześledźmy konstrukcję kodu źródłowego w przykładzie 1.3. Kod składa się z dwóch klas: RycerzJedi i Jedi1. Klasa RyczerzJedi nie posiada metody main, a więc nie jest główną klasą aplikacji. Klasa ta jest wykorzystywana dla celów danej aplikacji i jest skonstruowana z pól, konstruktora i metody. Pola oznaczają pewną własność obiektów jakie będą tworzone: nazwa - łańcuch znaków oznaczający nazwę rycerza Jedi; kolor_miecza - łańcuch znaków oznaczający kolor miecza świetlnego rycerza Jedi. Konstruktor jest metodą wywoływaną automatycznie przy tworzeniu obiektu. Stosuje się go do podawania argumentów obiektowi, oraz do potrzebnej z punktu widzenia danej klasy grupy operacji startowych. Wywołanie konstruktora powoduje zwrócenie referencji do obiektu danej klasy. Nie można więc deklarować konstruktora z typem void. W rozpatrywanym przykładzie konstruktor posiada dwa argumenty o nazwach takich samych jak nazwy pól. Oczywiście nie jest konieczne stosowanie takich samych oznaczeń. Jednak w celach edukacyjnych zastosowano tu te same oznaczenia, żeby pokazać rolę słowa kluczowego „this". Słowo „this" oznacza obiekt klasy w ciele której pojawia się „this". Stworzenie więc przypisania this.nazwa=nazwa powoduje przypisanie zmiennej lokalnej „nazwa" do pola danej klasy „nazwa". Klasa RycerzJedi ma również zdefiniowaną metodę opis, która wyświetla własności obiektu

czyli treść pól. Druga klasa Jedi1 zawiera metodę main(), a więc jest główną klasą aplikacji. W metodzie main() zawarto linie inicjowania obiektów np.:

RycerzJedi luke = new RycerzJedi(”Luke”, ”niebieski”);

Inicjowanie to polega na stworzeniu obiektu danej klasy, poprzez użycie słowa kluczowego „new” i wywołanie konstruktora klasy z właściwymi argumentami. W podanym przykładzie tworzony jest obiekt klasy RycerzJedi o nazwie luke i przypisuje się mu dane właściwości. Po stworzeniu dwóch obiektów w klasie Jedi1 następuje wywołanie metod *opis()* dla danych obiektów. Tak stworzona konstrukcja aplikacji umożliwia w sposób prosty skorzystanie z klasy RycerzJedi dowolnej innej klasie, aplikacji czy w aplecie.

1.3 Aplety

Oprócz aplikacji możliwe jest wywołanie określonego działania poprzez aplet. Aplet jest formą aplikacji wywoływanej w ściśle określonym środowisku. Aplet nie jest wywoływany wprost przez kod klasy *.class lecz poprzez plik HTML w kodzie którego zawarto odniesienie do kodu apletu *.class, np.:

<applet code=Jedi2.class width=200 height=100>
</applet>

Zapis ten oznacza, że w oknie o szerokości 200 i wysokości 100 będzie uruchomiony aplet o kodzie Jedi2.class. Zasadniczo aplet jest programem graficznym, stąd też wyświetlany tekst musi być rysowany graficznie. Polecenie:

System.out.println(”Rycerz Luke ma niebieski miecz.”);

nie spowoduje wyświetlenia tekstu w oknie apletu, lecz wyświetli tekst w konsoli Javy jeśli do takiej mamy dostęp. Uruchamiając aplet należy mieć Beta-kod Javy oraz plik odwoławczy w HTML.

Tworząc aplet tworzymy klasę dziedziczącą z klasy *Applet*, wykorzystując podstawowe metody takie jak *init(), start(), paint(), stop()* i *destroy().* Wywołując aplikację wywołujemy metodę *main()*, wywołując natomiast aplet wywołujemy przedstawione wyżej metody w podanej kolejności. Metody *init() i destroy()* są wykonywane jednorazowo (można uznać metodę *init()* za konstruktor apletu). Metody *start(), paint(), stop()* mogą być wykonywane wielokrotnie. Ponieważ aplet korzysta z klasy *Applet* oraz metod graficznych konieczne jest importowanie określonych pakietów. Rozpatrzmy następujący przykład:

Przykład 1.4:

```
// Jedi2.java :

import java.applet.Applet;
import java.awt.*;

public class Jedi2 extends Applet{
```

```
        public void paint(Graphics g){
                g.drawString(”Rycerz Luke ma niebieski miecz.„, 15,15);
        }

} // koniec public class Jedi2.class extends Applet
```

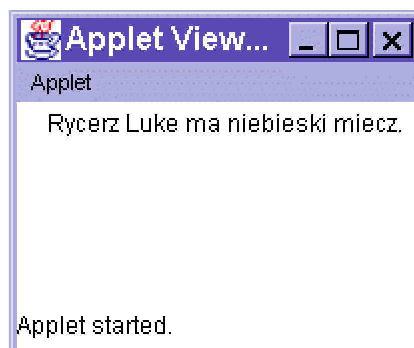
Jedi2.html *:*

```
<html>
<applet code=Jedi2.class width=200 height=100>
</applet>
</html>
```

Kompilacja kodu Jedi2.java odbywa się tak jak dla kodu aplikacji. Uruchomienie natomiast apletu wymaga otwarcia kodu HTML w przeglądarce WWW zgodnej z Javą lub za pomocą programu appletviewer dostarczanego w Java 2 SDK. Uruchomienie apletu odbywa się wówczas przez podanie:

appletviewer Jedi2.html

Pojawia się wówczas okno apletu o podanych w pliku Jedi2.html rozmiarach z napisem generowanym przez metodę *drawString()*.



Rysunek 2. Okno apletu dla przykładu 1.4.

Kod apletu z przykładu 1.4 zawiera jedynie implementację metody *paint()*, wywołanie której generuje kontekst graficzny urządzenia *Graphics g,* dzięki któremu możliwe jest użycie metody graficznej *drawString()*.
Ponieważ aplety są tylko specyficzną formą użycia klas dlatego apletami zajmiemy się w dalszej części tego opracowania.

1.4 Aplikacja i aplet w jednym kodzie

Przykład 1.5 obrazuje możliwość stworzenia w jednym kodzie źródłowym zarówno aplikacji jak i apletu. W zależności od metody interpretacji Beta-kodu (np. wykorzystanie „java" lub „appletviewer") otrzymamy zaprogramowany efekt.

Przykład 1.5:

```
// JediW.java:

import java.applet.Applet;
import java.awt.*;


class RycerzJedi{

        //pola
        String nazwa;
        String kolor_miecza;

        //konstruktor
        RycerzJedi(String nazwa, String kolor_miecza){
                this.nazwa=nazwa;
                this.kolor_miecza=kolor_miecza;
        }

        //metody
        void opis(){
                System.out.println(”Rycerz ”+nazwa+ ” ma ”+kolor_miecza+” miecz.”);
        }

}// koniec class RycerzJedi

public class JediW extends Applet {

        public void paint(Graphics g){
                g.drawString(”Rycerz Luke ma niebieski miecz.„, 15,15);
        }
        public void init(){
                RycerzJedi luke = new RycerzJedi(”Luke”, ”niebieski”);
                RycerzJedi ben = new RycerzJedi(”Obi-wan”,”biały”);
                luke.opis();
                ben.opis();
        }

        public static void main(String args[]){
                JediW j = new JediW();
                j.init();
        }// koniec public static void main(String args[])

}// koniec public class JediW
```

JediW.html:

```
<html>
<applet code=JediW.class width=200 height=100>
</applet>
</html>
```

W celu stworzenia uniwersalnego kodu należy zadeklarować główną klasę publiczną jako klasę dziedziczącą z klasy Applet, a następnie zdefiniować podstawowe metody apletu (*init(), paint()*) i aplikacji (*main()*).

1.5 Interpretacja, kompilacja i obsługa klas w Javie

Wywołanie programu stworzonego w Javie i obserwacja efektów jego działania jest dziedziną użytkownika programu. Programista powinien również wiedzieć jaka jest metoda powstawania kodu maszynowego danej platformy czyli jak jest droga pomiędzy kodem źródłowym a kodem wykonywanym. W sposób celowy użyto to sformułowania „kod wykonywany” zamiast „kod wykonywalny” aby ukazać, że nie zawsze użytkownik programu ma do dyspozycji kod wykonywalny na daną platformę, lecz czasami kod taki jest generowany w trakcie uruchamiania programu (np. wykonywanie programów stworzonych w Javie). W celu wyjaśnienia mechanizmu generacji kodu maszynowego danej platformy dla programów stworzonych w Javie przedstawić trzeba najpierw kilka podstawowych zagadnień związanych z kodem maszynowym.
Jak wiadomo kod maszynowy jest serią liczb interpretowaną przez komputer (procesor) w celu wywołania pożądanego efektu. Posługiwanie się ciągiem liczb w celu wywołania określonego działania nie jest jednak zbyt efektywne, a na pewno nie jest przyjazne użytkownikowi. Opracowano więc prosty język, który zawiera proste instrukcje mnemoniczne np. MOV, LDA, wywoływane z odpowiednimi wartościami lub parametrami. Przygotowany w ten sposób kod jest tłumaczony przez komputer na kod maszynowy. Język, o którym tu mowa to oczywiście asembler (od angielskiego assembly - gromadzić, składać). Wyrażenia napisane w asemblerze są tłumaczone na odpowiadające im ciągi liczb kodu maszynowego. Oznacza to, że jedna linia kodu asemblera (wyrażenie) generuje jedną linie ciągu liczb np. LDA A, J jest tłumaczone na 1378 00 1000, o ile kod LDA to 1378, akumulator to 00 a zmienna J jest pod adresem J. Kolejnym krokiem w stronę programisty było stworzenie języków wysokiego rzędu jak np. C, dla których pojedyncza linia kodu źródłowego tego języka może być zamieniona na klika linia kodu maszynowego danej platformy. Proces konwersji kodu źródłowego języka wysokiego poziomu do kodu maszynowego (wykonywalnego) danej platformy nazwano kompilacją (statyczną). Kompilacja składa się z trzech podstawowych procesów: tłumaczenia kodu źródłowego, generacji kodu maszynowego i optymalizacji kodu maszynowego. Tłumaczenie kodu źródłowego polega na wydobyciu z tekstu źródłowego programu elementów języka np. „if”, „while”, „(„, „class”; a następnie ich łączeniu w wyrażenia języka. Jeżeli napotkane zostaną niezrozumiałe elementy języka lub wyrażenia, nie będą zgodne z wzorcami danego języka, to kompilator zgłasza błąd kompilacji. Po poprawnym tłumaczeniu kodu źródłowego wyrażenia podlegają konwersji na kod maszynowy (czasem na kod asemblera, w sumie na to samo wychodzi). Następnie następuje proces optymalizacji całego powstałego kodu maszynowego. Optymalizacja ma za zadanie zmniejszenie wielkości kodu, poprawę szybkości jego działania, itp. Wyrażenia języka są często kompilowane, tworząc biblioteki, a więc gotowe zbiory kodów możliwe do wykorzystania przy konstrukcji własnego programu. Kompilując program (łącząc kody - linker) korzystamy z gotowych, pre-kompilowanych kodów. Biblioteki stanowią zarówno konieczną część zasobów języka programowania jak i mogą być wytwarzane przez użytkowników środowiska

tworzenia programów. Podsumowując kompilacja statyczna jest procesem konwersji kodu źródłowego na kod wykonywalny dla danej platformy sprzętowej.
Kolejną metodą konwersji kodu źródłowego na kod maszynowy jest interpretowanie kodu. Interpretowanie polega na cyklicznym (pętla) pobieraniu instrukcji języka, tłumaczeniu instrukcji, generacji i wykonywaniu kodu. Przykładowe interpretatory to wszelkie powłoki systemów operacyjnych tzw. shelle (DOS, sh, csh, itp.). Interpretowanie kodu ma więc tą wadę, że nie można wykonac optymalizacji kodu, gdyż nie jest on dostępny.
Nieco inaczej wygląda interpretowanie kodu źródłowego w Javie. Ponieważ zakłada się, że tworzone programy w Javie mogą być uruchamiane na dowolnej platformie sprzętowej, dlatego konieczne jest stworzenie takich interpretatorów, które umożliwią konwersję tego samego kodu źródłowego na określone i takie samo działanie niezależnie od platformy. Interpretowanie kodu jest jednak procesem czasochłonnym, tak więc konieczne są pewne modyfikacje w procesie interpretacji, aby uruchamialny program był efektywny czasowo. W przetwarzaniu kodu źródłowego Javy wprowadzono więc pewne zabiegi zwiększające efektywność pracy z programem stworzonym w Javie. Obsługa kodu źródłowego Javy przebiega dwuetapowo. Po pierwsze wykonywana jest kompilacja kodu źródłowego do kodu pośredniego zwanego kodem maszyny wirtualnej. Kod pośredni jest efektem tłumaczenia kodu źródłowego zgodnie z strukturą języka Java. Powstaje więc pewien zestaw bajtów, który aby mógł być uruchomiony musi być przekonwertowany na kod wykonywalny danej platformy. Zestaw bajtów wygenerowany poprzez użycie kompilatora „JAVAC” nosi nazwę kodu bajtów, lub B-kodu albo Beta-kodu. Wygenerowany B-kod jest interpretowany przez interpreter maszyny wirtualnej na kod wynikowy danej platformy. W celu dalszej poprawy szybkości działania programów opracowano zamiast interpretatora B-kod różne kompilatory dynamiczne. Kompilatory dynamiczne kompilują w locie Beta-kod do kodu wykonywalnego danej platformy. Stworzony w ten sposób kod wykonywalny jest umieszczany w pamięci komputera nie jest więc zapisywany w formie pliku (programu) na dysku. Oznacza to, że po skończeniu działania programu kod wykonywalny jest niedostępny. Ta specyficzna kompilacja dynamiczna jest realizowana przez tzw. kompilatory Just-In-Time (JIT). Najbardziej popularne kompilatory tej klasy, a zarazem umieszczane standardowo w dystrybucji JDK i JRE przez SUNa to kompilatory firmy Symantec. Używając programu maszyny wirtualnej „java” firmy SUN standardowo korzystamy z kompilatora dynamicznego JIT firmy Symantec zapisanego w pliku /jre/bin/symcjit.dll (dla Win95/98/NT). Można wywołać interpretator bez kompilacji dynamicznej poprzez ustawienie zmiennej środowiska wywołując przykładowo:

```
java -Djava.compiler=NONE JediW
```

gdzie JAVA.COMPILER jest zmienną środowiska ustawianą albo na brak kompilatora (czyli korzystamy z interpretatora) lub na dany kompilator.

Dalszą poprawę efektywności czasowej wykonywalnych programów Javy niesie ze sobą łączona interpretacja i kompilacja dynamiczna B-kodu. Jest to wykonywane przez najnowszy produkt SUN-a: Java HotSpot. Rozwiązanie to umożliwia kontrolowanie efektywności czasowej interpretowanych metod i w chwili gdy określona metoda jest wykryta jako czasochłonna wówczas generowany jest dla niej kod poprzez kompilator dynamiczny. Odejście od całkowitej kompilacji dynamicznej

kodu jest powodowane tym, że kod wykonywalny zajmuje dużo miejsca w pamięci i jest mało efektywny w zarządzaniu.

Standardowo dla potrzeb tego kursu używany będzie jednak kompilator JIT firmy Symantec wbudowany w dystrybucję JDK w wersji 1.2.

W celu zrozumienia metod tłumaczenia i interpretacji (kompilacji) kodu w Javie warto prześledzić proces ładowania klas do pamięci. Okazuje się, że dla każdej klasy stworzonej przez użytkownika generowany jest automatycznie obiekt klasy *Class*. W czasie wykonywania kodu, kiedy powstać ma obiekt stworzonej klasy maszyna wirtualna sprawdza, czy został już wcześniej załadowany do pamięci obiekt klasy *Class* związany z klasą dla której ma powstać nowy obiekt. Jeśli nie został załadowany do pamięci odpowiedni obiekt klasy Class wówczas maszyna wirtualna lokalizuje i ładuje B-kod klasy źródłowej **.class* obiektu, który ma powstać w danym momencie wykonywania programu. Oznacza to, że tylko potrzebny B-kod jest ładowany przez maszynę wirtualną (ClassLoader). Jeśli w danym wykorzystaniu programu nie ma potrzeby wykorzystania określonych klas, to odpowiadającym im kod nie jest ładowany.

W celu zobrazowania zagadnienia rozpatrzmy następujący program przykładowy:

Przykład 1. 6:

```
// Rycerze.java:

class Luke {
  Luke (){
    System.out.println("Moc jest z toba Luke!");
  }
  static {
    System.out.println("Jest Luke!");
  }
}// koniec class Luke

class  Anakin{
  Anakin (){
    System.out.println("Nie lekcewaz ciemnej strony mocy Anakin!");
  }
  static {
    System.out.println("Jest Anakin!");
  }
}// koniec class Anakin

class Vader {
  Vader (){
    System.out.println("Ciemna strona mocy jest potezna!!!");
  }
  static {
    System.out.println("Jest Vader!");
  }
}//// koniec class Vader
```

```
public class Rycerze {
  public static void main(String args[]) {
    System.out.println("Zaczynamy. Kto jest gotow pojsc z nami?");
    System.out.println("Luke ?");
    new Luke();
    System.out.println("Witaj Luke!");

    System.out.println("Anakin ?");
    try {
      Class c = Class.forName("Anakin");
      /* usunięcie komentarza w kolejnej linii spowoduje utworzenie obiektu
         klasy Anakin, a więc wywołany zostanie również konstruktor */
      // c.newInstance();
    } catch(Exception e) {
      e.printStackTrace();
    }
    System.out.println("Witaj Anakin!");
    System.out.println("Ktos jeszcze ?");
    new Vader();
    System.out.println("Gin Vader!");
  }
} // public class Rycerze
```

W przykładzie tym zdefiniowano trzy klasy *Luke*, *Anakin* oraz *Vader*, w których ciałach umieszczono kod statyczny (wykonywany wraz w ładowaniem klasy) generujący napis na konsoli. Dodatkowo, klasy te posiadają konstruktory wykonywane gdy wywoływany jest obiekt danej klasy. Klasa głowna aplikacji *Rycerze* wywołuje szereg komunikatów powiązanych z wykorzystaniem klas *Luke*, *Anakin* oraz *Vader*. Początkowo wywoływany jest komunikat rozpoczęcia wykonywania głównej metody aplikacji (*"Zaczynamy. Kto jest gotow pojsc z nami?")*. Po stworzeniu obiektu klasy Luke zostanie załadowana ta klasa, uruchomiona część statyczna klasy oraz wywołany konstruktor tej klasy. Oznacza to, że pojawią się da komunikaty: *„Jest Luke!”* oraz *„Moc jest z toba Luke!”*. Odwołanie się do kolejnej klasy *Anakin* jest zgoła odmienne. Nie tworzony jest obiekt tej klasy, lecz tworzony jest obiekt klasy *Class* skojarzony z klasą o podanej nazwie czyli *Anakin*. Klasa Anaikn jest więc ładowana do systemu co objawia się wyświetleniem komunikatu: *"Jest Anakin!"*. Nie zostanie natomiast wywołany konstruktor tej klasy, gdyż nie tworzymy obiektu tej klasy. Można to zrobić usuwając komentarz przy instrukcji *c.newInstance();* powodującej tworzenie nowego wystąpienia klasy *Anakin* (konwersja klas *Class* - > *Anakin*) czyli tworzenie obiektu tej klasy. Obsługa klasy *Vader* jest podobna do obsługi klasy *Luke*. Kompilując powyższy kod oraz wywołując go z interpretatorem lub kompilatorem dynamicznym Symantec JIT dla wersji JDK 1.2 (Symantec Java! Just-In-Time compiler, version 3.00.078(x) for JDK 1.2) otrzymamy:

Zaczynamy. Kto jest gotow pojsc z nami?
Luke ?
Jest Luke!
Moc jest z toba Luke!
Witaj Luke!
Anakin ?

Jest Anakin!
Witaj Anakin!
Ktos jeszcze ?
Jest Vader!
Ciemna strona mocy jest potezna!!!
Gin Vader!

lub gdy tworzymy obiekt przez *c.newInstance()*:

Zaczynamy. Kto jest gotow pojsc z nami?
Luke ?
Jest Luke!
Moc jest z toba Luke!
Witaj Luke!
Anakin ?
Jest Anakin!
Nie lekcewaz ciemnej strony mocy Anakin!
Witaj Anakin!
Ktos jeszcze ?
Jest Vader!
Ciemna strona mocy jest potezna!!!
Gin Vader!

Ciekawe jest to, że w niektórych wersjach interpretacji czy kompilacji wynik będzie zupełnie inny na przykład dla kompilatora *Symantec Java! Just-In-Time compiler, version 210-054 for JDK 1.1.2*:

Jest Luke!
Jest Vader!
Zaczynamy. Kto jest gotow pojsc z nami?
Luke ?
Moc jest z toba Luke!
Witaj Luke!
Anakin ?
Jest Anakin!
Nie lekcewaz ciemnej strony mocy Anakin!
Witaj Anakin!
Ktos jeszcze ?
Ciemna strona mocy jest potezna!!!
Gin Vader!

Na tym przykładzie widać jasno, że najpierw analizowany był cały kod i załadowane zostały te klasy, których obiekty są wywoływane w metodzie *main()*, czyli klasy *Luke* i *Vader*.

1.6 Wyjątki

W kodzie programu z przykładu 1.6 zostały wykorzystane instrukcje *try* i *catch*:

```
try {
    Class c = Class.forName("Anakin");
    /* usunięcie komentarza w kolejnej linii spowoduje utworzenie obiektu
       klasy Anakin, a więc wywołany zostanie również konstruktor */
    // c.newInstance();
} catch(ClassNotFoundException e) {
    e.printStackTrace();
}
```

Otóż w Javie oprócz błędów mogą również występować wyjątki. Wyjątki są to określone sytuacje niewłaściwe ze względu na funkcjonowanie klas lub metod. Przykładowe wyjątki to np. dzielenie przez zero, brak hosta o podanym adresie, brak pliku o podanej ścieżce, czy brak klasy. Każdy wyjątek związany jest z określoną klasą i jej metodami. Jeżeli dana metoda może spowodować wystąpienie wyjątku (co jest opisane w dokumentacji Javy) to należy wykonać pewne działanie związane z obsługa wyjątku. Metodę taką zamyka się wówczas w bloku kodu oznaczonego instrukcją warunkową *try* (spróbuj). Blok należy zakończyć działaniem przewidzianym dla sytuacji wyjątkowej w zależności od powstałego wyjątku. Detekcja rodzaju wyjątku odbywa się poprzez umieszczenie instrukcji catch (nazwa wyjątku i jego zmienna), w bloku której definiuje się działanie (obsługę wyjątku). Przykładowe działanie to wyświetlenie komunikatu na konsoli platformy. Wystąpienie wyjątku i jego właściwa obsługa nie powoduje przerwania pracy programu. Nazwy i typy wyjątków są zdefiniowane wraz z klasami i interfejsami w odpowiednich pakietach w dokumentacji Javy. Wyjątek jest definiowany przez właściwą mu klasę o specyficznej nazwie zawierającej fraze *Exception*. Przykłądowe klasy wyjątków to:

w pakiecie java.io.*:

EOFException - koniec pliku
FileNotFoundException - brak pliku
InterruptedIOException - przerwanie operacji we/wy
IOException - klasa nadrzędna wyjątków we/wy

w pakiecie java.lang.*:

ArithmeticException - wyjątek operacji arytmetycznych np. dzielenie przez zero,
ArrayIndexOutOfBoundsException - przekroczenie zakresu tablicy,
ClassNotFoundException - brak klasy,
Exception - klasa nadrzędna wyjątków.

Przykładowe błędy:

NoSuchFieldError - błąd braku danego pola w klasie,
NoSuchMethodError - błąd braku danej metody w klasie,
OutOfMemoryError- błąd braku pamięci.

Niektóre wyjątki dziedziczą z klasy RuntimeException, czyli są to wyjątki, które powstają dopiero w czasie działania programu i nie można ich przewidzieć, np. związać z określoną metodą. Przykładowym wyjątkiem tego typu jest *ArithmeticException* - wyjątek operacji arytmetycznych np. dzielenie przez zero. Dla

wyjątków klasy RuntimeException nie jest wymagane stosowanie obsługi wyjątków (tzn. kompilator nie zwróci błędu). Programista może jednak przewidzieć (powinien przeiwdzieć) wystąpienie tego typu wyjątku i obsłużyć go w kodzie programu.
Istnieje jeszcze jedna możliwość deklaracji, że dana procedura może zwrócić wyjątek. Zamiast stosować przechwytywanie wyjątku deklaruje się kod poprzez użycie słowa kluczowego *throws* po nazwie metody, w której ciele występują metody powodujące możliwość powstania sytuacji wyjątkowej np.:

```
public class Rycerze {
  public static void main(String args[]) throws Exception{
(..)
    Class c = Class.forName("Anakin");
(..)
  }
} // public class Rycerze
```

W powyższym przykładzie oznaczenie metody main() jako metody zwracającej jakiś wyjątek (ponieważ *Exception* jest nadklasą klas wyjątków) zwalnia nas (w sensie nie będzie błędu kompilacji) od używania instrukcji obsługi wyjątków. Zamiast klasy *Exception* można użyć konkretnego wyjątku jaki jest związany z obsługą kodu wewnątrz metody np. *ClassNotFoundException*.

1.7 Klasy wewnętrzne

W powyższych przykładach zdefiniowano klasy pomocnicze poza ciałem klasy głównej programu (poza klasą publiczną). Klasy tak zdefiniowane nazywa się klasami zewnętrznymi (outer classes). W Javie można również definiować klasy w ciele klasy publicznej czyli tworzyć klasy wewnętrzne (inner classes).

Przykładowy kod zawierający klasę wewnętrzną może wyglądać następująco:

Przykład 1.7:

```
// MasterJedi.java:

public class MasterJedi {
 public long pesel;

 public MasterJedi(long i) {
   pesel = i;
 }

 class StudentJedi {
  public long pesel;

  public StudentJedi(String s, long j)  {
   System.out.println(s+j);
  };
 }//koniec class StudentJedi
```

```
 public static void main(String args[]) {
   MasterJedi mj = new MasterJedi(90100903890L);
   System.out.println("PESEL mistrza to: " + mj.pesel);
   MasterJedi.StudentJedi sj= mj.new StudentJedi("PESEL +
   studenta to: ", 80081204591L);
 }
}//koniec public class MasterJedi
```

W metodzie main() przykładu 1.7 tworzony jest obiekt klasy wewnętrznej StudentJedi. Stworzenie obiektu tej klasy jest możliwe tylko wtedy, gdy istnieje obiekt klasy zewnętrznej, w tym przypadku klasy MasterJedi. Konieczne jest więc najpierw wywołanie obiektu klasy zewnętrznej, a później poprzez odwołanie się do tego obiektu stworzenie obiektu klasy wewnętrznej.
Stosowanie klasy wewnętrznych oprócz możliwości grupowania kodu i sterowania dostępem daje inne możliwości, do których powrócimy w dalszych rozważaniach na temat programowania obiektowego, a w szczególności na temat interfejsów.

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.

ASCII/8859-1 Text

A	0100 0001
S	0101 0011
C	0100 0011
I	0100 1001
I	0100 1001
/	0010 1111
8	0011 1000
8	0011 1000
5	0011 0101
9	0011 1001
-	0010 1101
1	0011 0001
	0010 0000
t	0111 0100
e	0110 0101
x	0111 1000
t	0111 0100

Unicode Text

A	0000 0000 0100 0001
S	0000 0000 0101 0011
C	0000 0000 0100 0011
I	0000 0000 0100 1001
I	0000 0000 0100 1001
	0000 0000 0010 0000
天	0101 1001 0010 1001
地	0101 0111 0011 0000
	0000 0000 0010 0000
س	0000 0110 0011 0011
ل	0000 0110 0100 0100
ا	0000 0110 0011 0111
م	0000 0110 0100 0101
	0000 0000 0010 0000
α	0000 0011 1011 0001
≰	0010 0010 0111 0000
γ	0000 0011 1011 0011

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 I; oznacza deklaracje uchwytu I 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: I = 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 = "śłąęóżźćń";
  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.

Rozdział 3 Programowanie obiektowe

3.1 Obiekty
3.2 Klasy abstrakcyjne
3.3 Interfejsy, specyfikatory dostępu
3.4 Statyczne klasy wewnętrzne
3.5 Klasy anonimowe
3.6 Adaptery
3.7 Dziedziczenie
3.8 Niszczenie obiektów – zwalnianie pamięci
3.9 Tablice

3.1 Obiekty

Człowiek różnymi metodami modeluje otaczający do świat, jego właściwości i procesy w nim zachodzące. W opisie rzeczywistości wprowadzono pojęcia kategoryzujące elementy świata. Do najbardziej znanych pojęć tego typu należy zaliczyć „byt” jako coś co istnieje. Starożytni filozofowie dokonywali różnych podziałów bytów. Wprowadzono podział na bytu ogólne i jednostkowe. Arystoteles wprowadził ciekawy opis bytu używając terminów forma i materia. Tak rozpoczęto klasyfikować byty pod względem ich form, czy też gatunku lub inaczej typu albo wreszcie klasy. Można więc przedstawić klasę takich bytów, które posiadają wspólne właściwości (formę). Klasa (forma) opisuje byt, jej połączenie z materią tworzy byt. Można więc powiedzieć, że materia połączona z określoną formą tworzy jednostkowy byt - czyli obiekt. W ten sposób uzyskano podstawę założeń programowania obiektowego: Obiekt jest jednostkowym wystąpieniem Klasy go opisującej. Oznacza to, że za pomocą programowania obiektowego tworzony jest model bytu, a nie jego opis ilościowy, tak jak to jest wykonywane w programowaniu proceduralnym. Opis obiektu w klasie odbywa się poprzez modelowanie jego zachowania (metody) i stanu (pola). Jak wspomniano już na początku tego opracowania może istnieć wiele obiektów danej klasy. Każdy z nich jest jednak jednostkowy i istnieje w pamięci komputera. Dostęp do obiektu jest możliwy za pomocą „uchwytu” (odwołanie do pamięci - stery - gdzie przechowywany jest obiekt). Nowy obiekt danej klasy tworzony jest za pomocą instrukcji new z podaniem nazwy klasy, np.:

```
Rycerz luke = new Rycerz();
```

oznacza, że tworzony jest nowy obiekt typu Rycerz, do którego przywiązany jest „uchwyt” luke. Tworzenie obiektu danej klasy bardzo dobrze ilustruje stosowany już w tej pracy kod obsługujący ładowanie klas:

```
Class c = Class.forName(„Rycerz”);
Rycerz luke = c.newInstance();
```

Kod ten wskazuje, że dla potrzeb tworzenia obiektów klasy Rycerz pobierany jest kod tej klasy a następnie tworzone jest nowe wystąpienie tej klasy. Oczywiście znacznie prostsze jest stosowanie instrukcji new.
Jeżeli temu samemu „uchwytowi” przypisany zostanie inny obiekt, wówczas obiekt, na który pierwotnie wskazywał „uchwyt” ginie. Nie ma więc potrzeby w Javie usuwania nieużywanych obiektów, gdyż jest to wykonywane automatycznie. Każdy

obiekt jest więc jednostkowym wystąpieniem klasy i ma charakter dynamiczny. Można jednak wyróżnić elementy niezmienne dla obiektów tej samej klasy. Przykładowo liczba Rycerzy nie jest własnością bytu (obiektu) lecz klasy, która opisuje byty tego typu. Oznacza to, konieczność definicji nie dynamicznych lecz statycznych (niezmiennych) pól i metod klasy, do których odwołanie odbywa się nie przez obiekty lecz przez nazwę klasy obiektów. Wszystkie pola i metody statyczne muszą być wyróżnialne poprzez oznacznik static. Określenie pola klasy jako static oznacza, że jego stan jest jednostkowy dla wszystkich obiektów danej klasy - jest własnością klasy a nie obiektów. Obiekt zmieniając stan pola statycznego zmienia go dla wszystkich innych obiektów. Przykładowo:

Przykład 3.1

```
//Republika.java

class Rycerz{
        static int liczbaRycerzy=0;
        int numerRycerza =0;
        Rycerz (String imie){
                System.out.println(„Rada Jedi głosi: „+ imie+” jest nowym Rycerzem Jedi”);
        }
}

public class Republika{

  public static void main (String args[]){

        Rycerz yoda = new Rycerz(„Yoda”);
        Rycerz anakin = new Rycerz(„Anakin”);
        Rycerz luke = new Rycerz(„Luke”);

        yoda.numerRycerza=1;
        yoda.liczbaRycerzy++;
        System.out.println(„Liczba rycerzy wg. Yody: „+yoda.liczbaRycerzy);
        System.out.println(„Liczba rycerzy wg. Anakina: „+anakin.liczbaRycerzy);
        System.out.println(„Liczba rycerzy wg. Luke’a: „+luke.liczbaRycerzy);

        anakin.numerRycerza=2;
        anakin.liczbaRycerzy++;
        System.out.println(„Liczba rycerzy wg. Yody: „+yoda.liczbaRycerzy);
        System.out.println(„Liczba rycerzy wg. Anakina: „+anakin.liczbaRycerzy);
        System.out.println(„Liczba rycerzy wg. Luke’a: „+luke.liczbaRycerzy);


        luke.numerRycerza=3;
        luke.liczbaRycerzy++;
        System.out.println(„Liczba rycerzy wg. Yody: „+yoda.liczbaRycerzy);
        System.out.println(„Liczba rycerzy wg. Anakina: „+anakin.liczbaRycerzy);
        System.out.println(„Liczba rycerzy wg. Luke’a: „+luke.liczbaRycerzy);
  }
}// koniec public class Republika
```

Powyższy przykład ukazuje brak zależności zmiennej liczbaRycerzy od poszczególnych obiektów. Statyczne pola klas są więc doskonałymi elementami przechowującymi informację o stanie klasy (np. ilość otwartych okien, plików, itp.). Możliwe jest również stworzenie statycznej metody, której działanie jest wywoływane bez konieczności tworzenia obiektu klasy, w ciele której zdefiniowano statyczną metodę. Przykładowa metoda statyczna może zwracać liczbę rycerzy (z przykładu 4.1) przechowywaną jako pole statyczne poprzez:

```
public static int liczbaR(){
	int lR=Rycerz.liczbaRycerzy;
	System.out.println(„Liczba Rycerzy Jedi w Republice wynosi: „+lR);
	return lR;
}
```

Należy pamiętać, co zostało już przedstawione wcześniej, tworzenie obiektu powoduje wywołanie procedury jego inicjowania zwanej konstruktorem. Konstruktor jest metodą o tej samej nazwie co nazwa klasy, dla której tworzony jest obiekt. Konstruktor jest wywoływany automatycznie przy tworzeniu obiektu. Stosuje się go do podawania argumentów obiektowi, oraz do potrzebnej z punktu widzenia danej klasy grupy operacji startowych. Wywołanie konstruktora powoduje zwrócenie referencji do obiektu danej klasy. Nie można więc deklarować konstruktora z typem void. Kod konstruktora może zawierać wywołanie innego konstruktora tej samej klasy lub wywołanie konstruktora nadklasy. Kolejność wołania konstruktorów w kodzie danego konstruktora jest następująca:

```
NazwaKlasy(argumety){
	this(argumenty1); //wywołanie innego konstruktora tej samej klasy
	super(argumenty1); //wywołanie konstruktora nadklasy
	kod;
}
```

Jeżeli programista jawnie nie zdefiniował konstruktora to jest on tworzony domyślnie, jako kod pusty bez argumentów , np.:

```
NazwaKlasy(){
}
```

Rozważania dotyczące klas zamieszczono na początku tego opracowania. W Javie istnieje możliwość sprawdzenia czy dany obiekt jest wystąpieniem danej klasy. Do tego celu służy operator instanceof, np. luke instanceof Rycerz. Efektem działania operatora jest wartość logiczna true - jeśli obiekt jest danej klasy, lub false - w przeciwnym przypadku.

3.2 Klasy abstrakcyjne

Kolejnym bardzo ważnym elementem programowania obiektowego jest odwołanie się do rozważań w filozofii nad możliwością istnienia bytów ogólnych (pojęć ogólnych, uniwersalii, itp.). Przykładowe pytanie tych rozważań może być następujące: Czy może istnieć obiekt ogólny Rycerz? Teoretycznie w Javie nie może

istnieć taki obiekt, lecz istnieje obiekt klasy Class związany i reprezentujący daną klasę (np.: Class c = Class.forName(„Rycerz”);). Obiekt taki jest wykorzystywany przez Javę do obsługi klas (szczególnie w zarządzaniu ładowaniem klas do pamięci). W Javie istnieje jeszcze jeden typ klas, dla których nie można inicjować obiektów metodą new. Ponieważ nie można stworzyć obiektów takiej klasy, klasa taka nie ma rzeczywistego wykorzystania w programowaniu obiektowym i jest oznaczana jako abstrakcyjna - abstract. Klasa abstrakcyjna zawiera w opisie obiektu również metody abstrakcyjne. W celu stworzenia obiektu, dla którego opis znajduje się w klasie abstrakcyjnej należy stworzyć klasę pochodną klasy abstrakcyjnej i podać rzeczywisty opis (realizację) wszystkim metodom abstrakcyjnym zdefiniowanym w klasie abstrakcyjnej. Klasy abstrakcyjne stosuje się wówczas, gdy na danym stopniu opisu ogólnego możliwych obiektów nie można podać rzeczywistej realizacji jednej lub wielu metod. Przykładowo tworząc klasę Statek nie można podać metody obliczającej pole powierzchni statku, gdyż ta zależy od danego typu statku kosmicznego, dla którego dany jest wzór na pole powierzchni. Tak więc klasa pochodna typu statku np.: GwiezdnyNiszczyciel, może zdefiniować w swym ciele metodę o tej samej nazwie co w klasie Statek, lecz określonej implementacji (realizacji). Dzięki temu wszystkie typy statków opisywane własnymi klasami będą miały podobny opis ogólny, ułatwiający wymianę informacji. Problem ten zilustrowano przykładem 3.2:

Przykład 3.2:

```
//Flota.java

abstract class Statek{
	int numerStatku;
	int liczbaDzial;
	long predkoscMax;
	public abstract int polePowierzchni();
	public void informacje(){
		System.out.println("Liczba dział = "+liczbaDzial);
		System.out.println("Prędkość maksymalna = "+predkoscMax);
		System.out.println("Numer identyfikacyjny = "+numerStatku);
	}

} // koniec abstract class Statek{

class GwiezdnyNiszczyciel extends Statek{
	int wysTrojkata;
	int dlgPodstawy;
	GwiezdnyNiszczyciel(int numer){
		numerStatku=numer;
	}
	public int polePowierzchni(){
		return (wysTrojkata*dlgPodstawy/2);
	}
}// koniec class GwiezdnyNiszczyciel

class GwiezdnySokol extends Statek{
	int szer;
	int dlg;
	GwiezdnySokol(int numer){
```

```
            numerStatku=numer;
        }
        public int polePowierzchni(){
            return (dlg*szer);
        }
} // koniec class GwiezdnySokol


public class Flota{
        public static void main(String args[]){
            GwiezdnyNiszczyciel gw1 = new GwiezdnyNiszczyciel(1);
            gw1.wysTrojkata=200;
            gw1.dlgPodstawy=500;
            gw1.liczbaDzial=6;
            gw1.predkoscMax=100;
            GwiezdnySokol gs1 = new GwiezdnySokol(1);
            gs1.dlg=40;
            gs1.szer=15;
            gs1.liczbaDzial=3;
            gs1.predkoscMax=120;

            gw1.informacje();
            System.out.println("Pole powierzchni Niszczyciela to: " + gw1.polePowierzchni() +" m(2)");

            gs1.informacje();
            System.out.println("Pole powierzchni Sokola to: " + gs1.polePowierzchni() +" m(2)");

        }
}// koniec public class Flota
```

W powyższym przykładzie określono 4 klasy wśród których jedna jest abstrakcyjną (Statek) w ciele której zawarto określone pola, metodę abstrakcyjną polePowierzchni() oraz metodę zdefiniowaną informacje(). Dwie klasy GwiezdnyNiszczyciel oraz GwiezdnySokol są klasami pochodnymi (dziedziczą – o czy później w tym rozdziale) klasy abstrakcyjnej i dokonują definicji metody abstrakcyjnej dziedziczonej klasy. Ostatnia klasa Flota zawiera wywołanie obiektów zdefiniowanych klas GwiezdnyNiszczyciel oraz GwiezdnySokol. Warto zauważyć, że następuje odniesienie do metody informacje() klasy abstrakcyjnej tak, jakby była to metoda obiektu klasy implementującej klasę abstrakcyjną (czyli GwiezdnyNiszczyciel albo GwiezdnySokol).

3.3 Interfejsy, specyfikatory dostępu

Klasa abstrakcyjna oprócz metod abstrakcyjnych ma również metody zdefiniowane. Abstrakcja wprowadzana przez taką klasę jest więc tylko częściowa. Można sobie oczywiście wyobrazić sytuację gdzie wszystkie metody będą abstrakcyjne. Wprowadzenie samych metod abstrakcyjnych, a więc tylko ich nazw, argumentów wywołania i zwracanych typów umożliwia stworzenie uniwersalnego zbioru funkcji możliwych do wykorzystania w różnych programach bez znajomości realizacji danej metody. Realizacja danej metody może być wykonywana w różny sposób w zależności od potrzeb bądź od otoczenia sprzętowo-programowego (np. odczyt z portu, kodowanie wiadomości, itp.). Zbiór metod abstrakcyjnych jest więc

swoistym interfejsem pomiędzy kodem użytkownika a zrealizowanymi metodami. Java określa nowy typ zbiorczy nazywając go „*intereface*”, który składa się ze zbioru metod abstrakcyjnych oraz ze zbioru statycznych (*static*) i stałych (*final*) pól. Tworząc nową klasę wykorzystującą opis metod podany w interfejsie implementuje (*implements*) się dany interfejs, tworząc definicję (realizację) każdej metody abstrakcyjnej interfejsu. Przykładowy interfejs oraz jego implementacja może przebiegać następująco:

Przykład 3.3:

```
// BronJedi.java

interface MieczJedi {
        static final String typ="świetlny";
        abstract void dzwiek();
        abstract float moc(int oslabienie);
} // koniec interface MieczJedi

class BronLukea implements MieczJedi{
        int dlugosc;
        int szerokosc;
        int mocGeneracji;
        BronLukea(int d, int s, int m){
          this.dlugosc=d;
          this.szerokosc=s;
          this.mocGeneracji=m;
        }
        public void dzwiek(){
          System.out.println("Dźwięk:");
          System.out.println("zzzzzzZZZZZZzzzzzzz");
        }
        public float moc(int oslabienie){
          float r = szerokosc / 2;
          float moc_miecza= (mocGeneracji / (dlugosc * r * r * 3.1456f) ) / oslabienie;
          System.out.println("Moc miecza wynosi: "+moc_miecza);
          return moc_miecza;
        }

}// koniec class BronLukea

class BronVadera implements MieczJedi{
        int dlugosc;
        int szerokosc;
        int mocGeneracji;
        BronVadera(int d, int s, int m){
          this.dlugosc=d;
          this.szerokosc=s;
          this.mocGeneracji=m;
        }
        public void dzwiek(){
          System.out.println("Dźwięk:");
          System.out.println("uuuuuuuuUUUUUUuuuuu");
        }
        public float moc(int oslabienie){
```

```
            float r = szerokosc / 2;
            float moc_miecza= (mocGeneracji / (dlugosc * r * r * 3.1456f) ) / oslabienie;
            System.out.println("Moc miecza wynosi: "+moc_miecza);
            return moc_miecza;
          }
          public void info(){
            System.out.println("Miecz to broń słabych Jedi. ");
          }

}// koniec class BronVadera

public class BronJedi {

          public static void main(String args[]){
            BronLukea bl =new BronLukea(70,5,100);
            BronVadera bv =new BronVadera(60,5,80);
            System.out.println("Miecz "+bl.typ+" Lukea:");
            bl.dzwiek();
            bl.moc(2);
            System.out.println("Miecz "+bv.typ+" Vadera:");
            bv.dzwiek();
            bv.moc(2);
            bv.info();
            System.out.println("Miecz Luke'a ma większą moc! Giń Vader !");

          }
}// koniec public class BronJedi
```

W powyższym przykładzie stworzono interfejs opisujący miecz. Do opisu tego elementu wykorzystano pole typu String określające rodzaj miecza oraz dwie metody abstrakcyjne dotyczące opisu dźwięku i mocy mieczy. Definicja pola zawiera elementy deklaracji static oraz final. Otóż każde pole interfejsu, nawet jeśli nie jest to jawnie przedstawione jest typu stałego (final) i statycznego (static). Wszystkie elementy interfejsu, a więc zarówno pola jaki i metody są domyślnie zadeklarowane jako publiczne (public). Można zadeklarować je również jako protected lecz nie można jako private. Czym są specyfikatory dostępu public, protected oraz private? Otóż oznaczają one sposób dostępu do elementów klasy w procesie komunikacji pomiędzy różnymi obiektami i klasami (hermetyzacja – odseparowanie składników). Specyfikator public (publiczny) określa dany element jako ogólnodostępny dla każdego kodu wykorzystującego dany element (np. dla programu klienta korzystającego z biblioteki, w której zdefiniowano dany element). W celu zabronienia innym klasom korzystania z określonych elementów należy je zadeklarować jako private (prywatne). Tak zadeklarowane elementy będą własnością prywatną klasy i jej metod. Można określić również dostęp do elementów tylko w pakiecie klas. Należy wówczas zadeklarować takie elementy jako protected. Wówczas tylko klasy pochodne (obiekty klas pochodnych) mogą korzystać z elementów oznaczonych jako protected, inne klasy nie mają dostępu. Natura interfejsów jest taka, że muszą być implementowane jeżeli celowym ma być ich stworzenie. Dlatego elementy interfejsów nie mogą być specyfikowane jako private. Ważne jest również to, że każdy element nie zadeklarowany jawnie specyfikatorem dostępu ma dostęp oznaczany jako „friendly”. Specyfikator taki oznacza dostęp do elementów jako publiczny dla wszystkich elementów tego samego pakietu, lecz jako prywatny poza nim. Wykorzystując interfejsy należy pamiętać, że stworzenie definicji (realizacji) metody

abstrakcyjnej interfejsu wymaga podania takiego samego specyfikatora dostępu jak dla deklaracji metody abstrakcyjnej w interfejsie. W ogromnej większości przypadków jest to dostęp typu public. W przykładzie 3.3 w deklaracji metod abstrakcyjnych interfejsu pominięto specyfikator. Oznacza to, że zostanie użyty domyślny specyfikator public. Definicja metod abstrakcyjnych w klasach implementujących interfejs musi zawierać jawnie specyfikator public przed podaniem zwracanego typu metody. Następujący kod:

```
(..)
abstract float moc(int oslabienie);
(..)

float moc (int oslabienie){
//realizacja
}
(..)
```

wygeneruje w czasie kompilacji błąd, ponieważ implementacja metody abstrakcyjnej jest postrzegana na zewnątrz jako private. Należy ją więc jawnie zadeklarować jako public:

```
public float moc (int oslabienie){
        //realizacja
}
```

Wykorzystywany w interfejsach specyfikator final również określa dostęp. Ogólnie element oznaczony jako final jest elementem o wartości niezmiennej (np. stała, stąd nie ma konieczności stosowania słowa kluczowego const) i musi być zainicjowany w czasie deklarowania. Specyfikator final może jednak służyć do innych celów niż tylko oznaczanie stałych. Możliwe jest oznaczenie „uchwytu” do obiektu jako final. Wówczas konieczne jest zainicjowanie obiektu wraz z deklaracją oraz konieczna jest świadomość, że dany „uchwyt” nie może wskazywać innego obiektu w danym kodzie programu. Przykładowy kod:

```
(..)
final Rycerz r = new Rycerz(„Luke”);
(..)
r = new Rycerz(„Vader”);
(..)
```

spowoduje błąd ponieważ wystąpiła próba zmiany referencji „uchwytu” obiektu r. „Uchwyt” raz oznaczony jako final nie może wskazywać innego obiektu, jednakże nie oznacza to, że obiekt nie może się zmieniać. Słowem kluczowym final można też oznaczać pola bez ich inicjowania (blank final) , niemniej należy takie inicjowanie przeprowadzić zanim dane pole zostanie wykorzystane (np. w konstruktorze klasy). Specyfikator final może być również stosowany w metodach klasy. Oznaczenie argumentu metody jako final określi, że nie można dokonywać zmian na tym argumencie w ciele metody. Oznaczenie metody jako final powoduje dwie konsekwencje: po pierwsze sprawia, że metoda nie może podlegać zmianom w procesie dziedziczenia klas, po drugie kompilator pracując z taką metodą wgrywa jej kod zamiast tworzyć referencje (call) do kodu tej metody (podobnie jak inline w C).

Oczywiście można również określić klasę jako final. Wówczas niemożliwe jest dziedziczenie (tworzenie klas pochodnych) z tak określonej klasy.
Z wykorzystywaniem interfejsów w Javie związane są jeszcze dwa zagadnienia. Pierwsze z nich dotyczy prezentowanych już klas wewnętrznych (klas anonimowe), drugie adapterów.

3.4 Statyczne klasy wewnętrzne

Omawiając klasy wewnętrzne warto rozważyć przypadek kiedy nie trzeba tworzyć obiektu klasy zewnętrznej aby stworzyć obiekt klasy wewnętrznej. Jest to możliwe wówczas, gdy klasa wewnętrzna będzie zadeklarowana jako statyczna.

Przykład 3.4:

```
//MieczL70Info.java

interface Dlugosc{
	public String wyswietl();
}// koniec interface Dlugosc

public class MieczL70Info {
	 public static int d=70;

	 static class Dlg implements Dlugosc{
	  private int y;
	  public Dlg(int j)  {
	   y=j;
	   System.out.println(wyswietl());
	  }
	  public String wyswietl(){
	   return (new String("Długość miecza L70 = "+y));
	  }
	}//koniec static class Dlg

	public static Dlugosc info(int i){
	  return (new Dlg(i));
	}

	public static void main(String args[]) {
		MieczL70Info.info(d);
	 }
}//koniec public class MieczL70Info
```

W powyższym przykładzie stworzenie obiektu klasy wewnętrznej odbyło się bez potrzeby tworzenia obiektu klasy wewnętrznej. Zdefiniowana klasa wewnętrzna Dlg jest określona jako static, implementuje interfejs Dlugosc definiując jego metodę wyswietl(). Wywołanie stworzenia obiektu odbywa się poprzez uruchomienie metody klasy zewnętrznej MieczL70Info typu Dlugosc (interfejs), a mianowicie metody info().

3.5 Klasy anonimowe

Z punktu widzenia zastosowań interfejsów o wiele ciekawsze są innego rodzaju definicje klas wewnętrznych. Rozpatrzmy kolejny przykład:

Przykład 3.5:

```
//Bateria.java

interface MocGeneracji{
	abstract public int moc();
}// koniec interface MocGeneracji


public class Bateria{
	int val;
	Bateria(int poziomEnergii){
		this.val=poziomEnergii;
		System.out.println("Stan baterii= "+this.val);
	}
	public MocGeneracji m_gen(){
		return new MocGeneracji() {
			public int moc(){
				return (val / 10);
			}
		}; //średnik kończy linię instrukcji w ciele metody m_gen()
	}
	public static void main(String args[]){
		Bateria b = new Bateria(40);
		MocGeneracji mg = b.m_gen();
		System.out.println("Maksymalna moc generacji= "+mg.moc());
	}
}//koniec public class Bateria
```

Przykład powyższy wprowadza ciekawą konstrukcję. W ciele klasy głównej aplikacji zdefiniowano metodę m_gen() typu MocGeneracji (interfejs). Metoda ta musi zwracać obiekt typu MocGeneracji czyli obiekt interfejsu! Obiekt zwracany jest poprzez klasyczne wyrażenie w instrukcji return: new MocGeneracji(). Ciekawe jednak jest to, co dzieje się bezpośrednio po instrukcji tworzenia obiektu. Otóż definiowana jest tam klasa, nie posiadająca swojej nazwy, stąd nazywana klasą anonimową. W ciele tej klasy wyróżnianej blokiem kodu (klamry) zawarto definicję metody dla zadeklarowanej metody abstrakcyjnej interfejsu, którego obiekt jest tworzony. Stworzony obiekt klasy anonimowej jest automatycznie rzutowany (new MocGeneracji()) na typ interfejsu. Konstruktor klasy anonimowej jest oczywiście domyślny (metoda pusta). W ciele metody main() aplikacji wykorzystano stworzoną klasę anonimową w ten sposób, że zainicjowany obiekt typu interfejs: MocGeneracji jest właściwie obiektem klasy anonimowej, stąd tylko „uchwyt" obiektu jest deklarowany jako „uchwyt" typu MocGeneracji i wskazuje on na obiekt klasy anonimowej. Dlatego możliwe jest wywołanie właściwości obiektu poprzez zastosowanie „uchwytu" interfejsu. Dzięki temu możliwe jest wyświetlenie, w ostatniej linii kodu tej przykładowej aplikacji, komunikatu zawierającego wartość zwracaną

przez metodę obiektu klasy anonimowej. Należy pamiętać, że definiowanie klasy anonimowej ma te same właściwości co definiowanie klasy jawnej. Dlatego istotne jest rozpatrzenie przy konstrukcji takich klas zastosowanych praw dostępu do pól i metod. Przykładowo umieszczenie zmiennej (bez specyfikatora final) w ciele metody, w której definiuje się klasę anonimową korzystającą z tej zmiennej spowoduje błąd kompilacji.

3.6 Adaptery

Korzystanie z interfejsów ma jednak czasem swoje złe strony. Otóż stosując interfejs zdefiniowany w jakimś pakiecie w opracowywanym kodzie programista musi zawsze dokonać definicji wszystkich metod zadeklarowanych w interfejsie. Jeśli metod tych jest dużo a programista kilkakrotnie korzysta z danego interfejsu wykorzystując zaledwie dwie metody, wówczas wiele linijek powstałego kodu to kod martwy (nieużyteczny). W takich przypadkach warto stworzyć klasę implementującą dany interfejs, tworząc w jej ciele puste metody odpowiadające metodą zadeklarowanym w interfejsie. Korzystając w programie z interfejsu, korzysta się wówczas z tak stworzonej klasy, stanowiącej niejako element adaptujący interfejs do potrzeb programisty. Klasa adaptująca nosi nazwę w Javie adaptera. Wykorzystanie adaptera w kodzie tworzonego programu możliwe jest dzięki właściwościom dziedziczenia (tworzona klasa dziedziczy z klasy adaptera, czyli korzysta z jej metod) oraz przesłaniania (realizacja metody jest wykonywana w kodzie programu jako przepisanie metody z klasy adaptera). Dziedziczenie i przesłanianie są omówione dalej. Poniższy przykład ukazuje tworzenie i wykorzystanie klasy adaptera.

Przykład 3.6:

```
//Robot.java

interface R2D2{
        String wyswietl();
        String pokaz();
        int policz(int a, int b);
        float generuj();
        double srednia();
}

abstract class R2D2Adapter implements R2D2{

        public String wyswietl(){
                        return "";
        }
        public String pokaz(){
                        return"";
        }
        public int policz(int a, int b){
                        return 0;
        }
        public float generuj(){
                        return 0.0f;
        }
```

```
        public double srednia(){
                        return 0.0d;
        }
}

public class Robot extends R2D2Adapter{

        public String wyswietl(){
                String s = "Tekst ten jest tworem adaptacji R2D2";
                System.out.println(s);
                return s;
        }

        public static void main(String args[]) {
                Robot r = new Robot();
                r.wyswietl();
         }
}//koniec public class Robot
```

3.7 Dziedziczenie

Nazwa dziedziczenie związana jest z procesem ewolucyjnym, w którym potomkowie posiadają pewne cechy rodziców. Podobne znaczenie tej nazwy jest używane w programowaniu obiektowym. Otóż zakładając, że każdy typ lub gatunek może mieć swój podgatunek, a więc zawężony i uszczegółowiony opis obiektów, to właściwości tych obiektów będą wynikały zarówno ze specyfikacji gatunku jak i podgatunku. Inaczej mówiąc, elementy klas nadrzędnych będą również elementami ich pochodnych (klas dziedziczących z klasy nadrzędnej). Przykładowo klasą nadrzędną może być klasa Rycerz, klasami z niej dziedziczącymi mogą być klasy Człowiek, Gungan, Kobieta, itp. W Javie określenie dziedziczenia odbywa się poprzez użycie słowa kluczowego extends (rozszerza). Przykładowa deklaracja:

```
class Kobieta extends Rycerz{
(...)
}
```

określa, że tworzona klasa Kobieta dziedziczy z klasy Rycerz. W Javie możliwe jest tylko dziedziczenie typu jeden-do-jednego, co oznacza, że klasa może dziedziczyć tylko z jednej klasy nadrzędnej. Ponieważ klasa nadrzędna może również dziedziczyć z jednej klasy dla niej nadrzędnej otrzymuje się specyficzne drzewo dziedziczenia w Javie. Nadrzędną klasą dla wszystkich klas w Javie jest klasa Object. Wszystkie klasy bezpośrednio lub pośrednio z niej dziedziczą, czyli klasa ta stanowi korzeń drzewa dziedziczenia. Ponieważ znajduje się ona w pakiecie java.lang.*, można powiedzieć, że wszystkie klasy będą korzystały z tego pakietu. W Javie możliwe jest więc dziedziczenie wielopoziomowe polegające na tym, że wiele klas może dziedziczyć z jednej (lecz nie odwrotnie).
Oczywiście istnieje możliwość sterowania sposobu wykorzystania elementów klasy nadrzędnej przez klasy pochodne. Do podstawowych metod tego typu sterowania należą: przeciążenie metod oraz przepisanie metod. Przeciążenie polega w programowaniu obiektowym na rozszerzeniu działania danej operacji (o tej samej nazwie) na innego typu argumenty. Przykładowo przeciążenie operatorów oznacza

stworzenie możliwości wykorzystywania np. znaku '+' do dodawania dwóch obiektów. Jak wspomniano wcześniej przeciążenie operatorów nie jest dostępne w Javie poza jedynym wyjątkiem dotyczącym dodawania obiektów typu String. Przeciążenie metod polega na definicji zbioru metod o tej samej nazwie lecz operujących na innych typach danych (argumentach). Przykładowo :

Przykład 3.7

```
//StanRepubliki.java

class Rep{
	int x = 36;
	String wiad="Stan Republiki: ";
	String typ=" światów.";
	void stan(String s){
		System.out.println(wiad+s+typ);
	}
	void stan(int i){
		System.out.println(wiad+i+typ);
	}
}// koniec class Rep

public class StanRepubliki extends Rep{

	public static void main(String args[]){
		StanRepubliki st = new StanRepubliki();
		st.stan(36);
		st.stan("trzydzieści sześć");
	}
}//koniec public class StanRepubliki
```

W powyższym przykładzie zastosowano przeciążenie metod stan() klasy nadrzędnej Rep. Pierwotna metoda stan() zdefiniowana jest dla argumentu typu String. Druga metoda stan() dokonuje przeciążenia metody pierwotnej o obsługę argumentu typu int. W ten sam sposób możliwe jest wyświetlenie różnych typów danych przez bardzo często używaną w tej pracy instrukcję System.out.println(). Otóż w klasie PrintStream (obiekt out) zdefiniowano szereg metod println() dla różnych typów argumentów. Dzięki temu bardzo wygodne jest stosowanie tej samej nazwy metody bez zbędnego rozważania nad typem argumentu.
Innym sposobem sterowania relacjami pomiędzy elementami klasy nadrzędnej i pochodnej jest przepisywanie lub inaczej przesłanianie metod. Przesłanianie polega na stworzeniu w klasie pochodnej nowej definicji dla metody istniejącej w klasie nadrzędnej. Stworzony obiekt w czasie odwołania się do danej metody podczas wykonywania programu wywołuje odpowiedni kod. Przywiązanie odpowiedniego kodu metody, a więc kodu na nowo zdefiniowanej metody, jest zadaniem obiektu w czasie wykonywania programu. Proces ten jest często nazywany przywiązaniem dynamicznym (dynamic binding), a konstrukcja kodu zawierająca różne definicje tak samo zadeklarowanej metody określana jest polimorfizmem. Poniższy przykład ukazuje sposób przesłaniania (polimorfizmu) metod i jest odniesiony do prezentowanego wyżej kodu programu.

Przykład 3. 8:

```
//StanRepubliki1.java

class Rep{
	int x = 36;
	String wiad="Stan Republiki: ";
	String typ=" światów.";
	void stan(String s){
		System.out.println(wiad+s+typ);
	}
	void stan(int i){
		System.out.println(wiad+i+typ);
	}
}// koniec class Rep

public class StanRepubliki1 extends Rep{

	void stan(int i){
		i--;
		System.out.println("Bez planety Naboo Republika skłąda się z "+i +
		" światów");
	}

	public static void main(String args[]){
		StanRepubliki1 st = new StanRepubliki1();
		st.stan(36);
		st.stan("trzydzieści sześć");
	}
}//koniec public class StanRepubliki1
```

Przykład ten jest rozwinięciem kodu StanRepubliki i zawiera dodatkową definicję metody stan() w ciele klasy pochodnej StanRepubliki1. Nowa definicja powoduje przesłanianie starej, stąd wywołanie polecenia st.stan(36) wywoła pojawienie się napisu:

Bez planety Naboo Republika składa się z 35 światów

zamiast:

Stan Republiki: 36 światów

Przeciążanie i przesłanianie metod są zatem niezwykle efektywnymi metodami w konstruowaniu funkcjonalności obiektów.

3.8 Niszczenie obiektów – zwalnianie pamięci

Bardzo istotnym zagadnieniem w rozważaniach nad pracą z obiektami jest problem niszczenia obiektów. O ile tworzenie obiektów i ich inicjowanie za pomocą konstruktora jest jawnie określone (instrukcja new), o tyle niszczenie obiektów i zwalnianie przydzielonych im zasobów jest niejawne. W Javie nie istnieje operator delete (C++) umożliwiający usunięcie obiekty, lub operator free (C) zwalniający

przydzieloną pamięć. Dlaczego? Dlatego, że wszystko w Javie dzieje się dynamicznie. Dynamicznie przydzielana jest pamięć obiektowi, dynamicznie jest więc też zwalniana. Owa dynamiczność określa nic innego jak to, że programista nie kontroluje dostępu do pamięci, tak więc nie wie gdzie Java umieszcza poszczególne obiekty. Obiekt będący w pamięci komputera jest dostępny dla użytkownika poprzez „uchwyt". „Uchwyt" jest więc odniesieniem do obiektu i sprawia, że obiekt jest określony. Brak odniesienia do obiektu sprawia, że obiekt jest niewykorzystywany, tak więc Java może go zniszczyć, a co za tym idzie zwolnić pamięć. Niszczeniem obiektów i zwalanianiem pamięci zajmuje się w Javie proces działający w tle noszący nazwę GarbageCollection (czyli kolekcjoner śmieci). Obiekty bez referencji są umieszczane w „śmietniku" a pamięć im przydzielona jest zwalniana najszybciej jak to jest możliwe (proces kolekcjonera śmieci posiada niski priorytet stąd zwrócenie przydzielonej pamięci do systemu nie koniecznie odbędzie się natychmiastowo). Nie wiadomo kiedy proces GarbageCollection dokonuje zwalniania pamięci i programista nie ma żadnej, bezpośredniej możliwości kontrolowania tego procesu.

Przykład 3.9:

```
//Rozmiar.java

class Rep{
	int x = 36;
	String wiad="Stan Republiki: ";
	String typ=" światów.";
	void stan(String s){
		System.out.println(wiad+s+typ);
	}
	void stan(int i){
		System.out.println(wiad+i+typ);
	}
}// koniec class Rep

public class Rozmiar extends Rep{

	void stan(int i){
	  i--;
	  System.out.println("Republika składa się z "+i +" światów");
	}

	public static void main(String args[]){
	  System.gc();
	  Thread.currentThread().yield();
	  long s1 = Runtime.getRuntime().freeMemory();
	  System.out.println("Test rozmiaru1: "+s1+ " lat św.(3)");
	  long s2 = Runtime.getRuntime().freeMemory();
	  System.out.println("Test rozmiaru2: "+s2+ " lat św.(3)");
	  long stan1 = Runtime.getRuntime().freeMemory();
	  System.out.println("Test rozmiaru3: "+stan1+ " lat św.(3) \n");

	  Rozmiar[] r = new Rozmiar[10000];
	  long stan2 = Runtime.getRuntime().freeMemory();
	  System.out.println("Zadeklarowano 10000 obiektów, rozmiar: "+stan2+" lat św.(3)");
```

```
    for ( int i =0; i < r.length; i++ )
        r[i] = new Rozmiar();
    long stan3 = Runtime.getRuntime().freeMemory();
    System.out.println("Zainicjowano 10000 obiektów, rozmiar: "+stan3+ " lat św.(3) \n");

    r=null;

    long stan4 = Runtime.getRuntime().freeMemory();
    System.out.println("Brak odwołania, rozmiar: "+stan4+ " lat św.(3)");
   /*
    System.gc();
    long stan5 = Runtime.getRuntime().freeMemory();
    System.out.println("Wywołano likwidację, rozmiar: "+stan5+ " lat św.(3)\n");
   */ //można usunąć ten komentarz i wywołać GC
    int[] liczbaSatelitówPlanet = new int[100000];
    long stan6 = Runtime.getRuntime().freeMemory();
    System.out.println("Daklaracja 100000 int, rozmiar: "+stan6+ " lat św.(3)");
    for ( int i =0; i < liczbaSatelitówPlanet.length; i++ )
        liczbaSatelitówPlanet[i] = 12;
    long stan7 = Runtime.getRuntime().freeMemory();
    System.out.println("Inicjacja 100000 int, rozmiar: "+stan7+ " lat św.(3) \n");

    liczbaSatelitówPlanet=null;
    long stan8 = Runtime.getRuntime().freeMemory();
    System.out.println("Brak odwołania, rozmiar: "+stan8+ " lat św.(3)");
    System.gc();
    long stan9 = Runtime.getRuntime().freeMemory();
    System.out.println("Wywołano likwidację, rozmiar: "+stan9+" lat św.(3)");


   }
}//koniec public class Rozmiar
```

Warto prześledzić działanie powyższego programu. Otóż na początku programu wywoływany jest jawnie proces GarbageCollection. Wywołanie to jest wezwaniem do systemu aby wykonał on operację niszczenia nie wykorzystywanych obiektów. W kolejnym kroku zostaje trzykrotnie pobrana i wyświetlona informacja dotycząca wielkości stery. Wyświetlone wartości mogą się nieznacznie wahać (pamięć jest obszarem dynamicznym platformy) i dla standardowej wielkości początkowej sterty równej 1MB wyniosą około 800kB. Kolejną operacją w programie jest deklaracja tablicy 10 000 obiektów typu Rep. W wyniku tej deklaracji zostaną umieszczone w pamięci „uchwyty" w liczbie 10 000, przy czym każdy uchwyt obiektu jest reprezentowany przez 4 bajty (platforma Windows NT) co oznacza, że wielkość pamięci zmaleje o 40 000 bajtów. Następnie wykonano w programie inicjacje zadeklarowanych obiektów klasy Rep, co również zmniejszyło pamięć o tyle ile jest potrzebne przez 10 000 obiektów klasy Rep. Kolejnym krokiem jest zmiana odniesienia „uchwytu". Brak odniesienia nie wpłynął jednak bezpośrednio na zmianę wielkości wolnej pamięci. Jest to związane z tym, że GarbageCollection nie zwalnia pamięci bezpośrednio po zmianie odniesienia (generacji martwego obiektu), lecz dopiero po pewnym czasie (nie wiadomo dokładnie ile ponieważ jest to proces dynamiczny). W przykładowym programie wykonano następnie operację stworzenia tablicy zmiennych typu podstawowego int o rozmiarze 100 000. Ponieważ definicja tablicy zmiennych typu podstawowego powoduje automatyczne wypełnienie tej

tablicy wartościami domyślnymi danego typu, dlatego pamięć zmniejszyła się o wielkość 100 000 * 4 B (int), czyli o 400 000B. Podstawienie własnych wartości do pól tablicy nie zmienia wielkości wolnej pamięci. W kolejnym kroku zmieniono odniesienie „uchwytu” obiektu tablicy na null, tworząc w ten sposób obiekt tablicowy obiektem martwym. Jak wspomniano wcześniej zmiana odniesienia nie musi wywołać natychmiastowego zwolnienia pamięci. W celu wymuszenia zwolnienia pamięci wykonano jawne przywołanie procesu GarbageCollection. Przywołanie to może dać efekt (lecz nie zawsze musi taki efekt wystąpić od razu, szczególnie dla obiektów innych niż tablicowe -> patrz komentarz w kodzie rozważanego przykładu) i zwolniona może zostać pamięć zajmowana poprzednio przez obiekt tablicowy oraz zbiór obiektów klasy Rep. Praktycznie programista nie ma możliwości uzyskania kontroli nad pracą procesu GarbageCollection (a więc brak jest synchronicznej kontroli nad procesem zwalniania pamięci).
Praca z Maszyną Wirtualną Javy umożliwia ustawienie parametrów związanych z procesem GarbageCollection oraz związanych z wielkością pamięci. I tak wywołanie Maszyny Wirtualnej z opcją -Xnoclassgc wyłącza proces GarbageCollection (np.: java -Xnoclassgc Rozmiar); wywołanie Maszyny Wirtualnej z opcją -XmsNNN, gdzie NNN jest liczbą całkowitą, powoduje ustawienie początkowej wielkości sterty na NNN bajtów (np.: java -Xms8000000 Rozmiar); wywołanie Maszyny Wirtualnej z opcją -XmxNNN, gdzie NNN jest liczbą całkowitą, powoduje ustawienie maksymalnej wielkości sterty na NNN bajtów (np.: java -Xmr8000000 Rozmiar).

Pamięć nie jest jednak jedynym zasobem jaki może wykorzystywać obiekt. Dlatego ważne jest czasem jawne zwracanie zasobów w czasie niszczenia obiektu. Przykładowo trzeba czasem zamknąć plik (strumień do pliku), który jest otwarty albo zniszczyć gniazdo dla połączeń w oparciu o TCP/IP. Z tych względów stworzono w Javie możliwość wykonania działania w czasie niszczenia obiektu. Jest to możliwe dzięki przesłonięciu metody finalize() (metoda bez argumentów i nic nie zwracająca void). Instrukcje umieszczone w ciele tej metody zostaną wykonane w czasie niszczenia obiektu klasy, w ciele której znajduje się metoda finalize().

3.9 Tablice

Tworzenie tablicy w Javie jest jednoznaczne tworzeniu obiektu typu Array zawierającego zbiór elementów zadeklarowanego typu. Przykładowo definicja int [] liczby = new int[10]; tworzy obiekt zawierający 10 pól, każde przechowujące wartość domyślną typu int. Standardowo dla stworzonej tablicy istnieje jej obiekt, który przechowuje w niejawnym polu długość tablicy (pole length). Przechowanie wartości w tablicy polega na wykorzystaniu instrukcji przypisania z podaniem numeru pola w tablicy: liczba[3] = 123. W Javie można oczywiści stworzyć tablicę dowolnych obiektów, a nie tylko tablicę zmiennych podstawowych typów danych. Podobnie jak inne języki programowania Java umożliwia tworzenie tablic wielopoziomowych. Każdy dodatkowy poziom jest zaznaczany dodatkowym zbiorem nawiasów w instrukcji definiującej obiekt lub przy dowolnym odwołaniu się do elementu wielopoziomowej tablicy. Najprostszą tablica wielopoziomową jest oczywiście macierz definiowana jako np.: int [][] liczby = new int[10][20]; czyli jest to macierz o wymiarach 10 na 20 przechowująca elementy typu int. Rozmiar wielopoziomowej tablicy można uzyskać posługując się kolejno polem length, np.: long rozmiar1 = liczny[].length; long rozmiar2 = liczby.length.

Przykład 3.10:

```
//Dane.java


public class Dane extends Rep{



	public static void main(String args[]){

		int stan[][] = new int [2][2];
		for (int i = 0; i<stan[0].length; i++){
			for (int j = 0; j<stan.length; j++){
				stan[i][j]=7;
				System.out.println("Liczba satelitów planety "+i+
				" w galaktyce "+j+"wynosi: "+stan[i][j]);
			}
		}
		long wiek[] = new long[3];
		wiek[0]=600000000L;
		wiek[1]=35000000L;

		System.out.println("Wiek planety 0 = "+wiek[0]+ " lat");
		System.out.println("Wiek planety 1 = "+wiek[1]+ " lat");
		System.out.println("Wiek planety 2 = "+wiek[2]+ " lat");


	}
}//koniec public class Dane
```

W powyższym przykładzie stworzono dwie tablice. Pierwsza z nich jest macierzą o rozmiarach 2x2, której inicjalizacji dokonano w pętli for. Warto zwrócić uwagę na sposób określania rozmiaru macierzy. Stosowana metoda jest bardzo efektywna i szybka. Druga tablica jest zdefiniowania do przechowywania trzech elementów typu long. Inicjowanie tablicy odbywa się tylko dla dwóch pól. Oznacza to, że wartość trzeciego pola będzie domyślna dla typu long czyli 0. Efekty stworzenia i zainicjowania tablic są wyświetlone na ekranie.

Rozdział 4 Programowanie współbieżne – wątki

4.1 Rys historyczny
4.2 Tworzenie wątków w Javie
4.3 Priorytety
4.4 Przetwarzanie współbieżne a równoległe
4.5 Przerywanie pracy wątkom
4.6 Przerywanie tymczasowe
4.7 Synchronizowanie wątków
4.8 Grupy wątków – ThreadGroup
4.9 Demony
Bibliografia

4.1 Rys historyczny

Na początku człowiek stworzył maszynę. Dalsza historia rozwoju technologii to próby uzyskania jak największej efektywności działania maszyny. Stwierdzenie to jest również słuszne dla komputera - maszyny matematycznej.
W pierwszym okresie użytkowania komputerów, celem zwiększenia ich efektywności działania, stosowano programowanie wsadowe. W podejściu takim, programy wykonywały się cyklicznie: jeden po drugim. Komputer stał się maszyną wielozadaniową wykonującą różne operacje, np. liczył całkę, po czym rozwiązywał model fizyczny zjawiska, następnie obliczał wymaganą grubość pancerza czołgu, itd. Ponieważ typy zadań są często różne, wymagają odmiennych zasobów komputera, dlatego opracowano rozwiązanie umożliwiające dzielenie zasobów komputera pomiędzy poszczególnych użytkowników. W ten sposób każdy użytkownik otrzymywał prawo wykonania swojego programu na komputerze. Rozwiązanie to było niezwykle istotne w przypadku konieczności wykonania dwóch zadań, skrajnie obciążającego czasowo komputer oraz mało obciążającego czasowo komputer, np. policzenie parametrów skomplikowanego (wielowymiarowego) modelu i rozwiązanie układu równań. Wielozadaniowość nie stanowiła jednak wystarczająco efektywnego rozwiązania. Zaproponowano więc możliwość podziału zadań (programów) na mniejsze funkcjonalnie spójne fragmenty kodu wykonywanego (odseparowane strumienie wykonywania działań) zwane wątkami (threads). Wątek jest więc swoistym podprogramem (zbiorem wykonywanych operacji) umożliwiającym jego realizację w oderwaniu od innych wątków danego zadania. Relacja pomiędzy wątkami określana głównie poprzez dwa mechanizmy: synchronizację i zasadę pierwszeństwa. Synchronizacja wątków jest niezwykle istotnym zagadnieniem, szczególnie wówczas gdy są one ze sobą powiązane, np. korzystają z wspólnego zbioru danych. Określanie zasad pierwszeństwa - priorytetów jest pomocne wówczas, gdy efekt działania danego wątku jest o wiele bardziej istotny dla użytkownika niż efekt działania innego wątku. Oczywiście wątki mogą być generowane nie tylko jawnie przez użytkownika komputera ale również niejawnie poprzez wywołane programy komputerowe. Przykładowo dla Maszyny Wirtualnej Javy działa co najmniej kilka wątków: jeden obsługujący kod z metodą *main*(), drugi obsługujący zarządzanie pamięcią (GarbageCollection) jeszcze inny zajmujący się odświeżaniem ekranu, itp. Tworzenie wielu wątków jest docelowo związane z możliwością ich równoczesnego wykonywania (programowanie równoległe). Możliwe jest to jednak jedynie w systemach wieloprocesorowych. W większości obecnych komputerach osobistych i stacjach roboczych współbieżność wątków jest emulowana. Stosowanie podziału kodu na liczne wątki (procesy danego programu-

zadania) ma więc charakter zwiększenia efektywności działania (głównie w systemach wieloprocesorowych) oraz ma charakter porządkowania kodu (głównie w systemach jednoprocesorowych - podział zadań).
Programy Javy (zarówno aplety jak i aplikacje) są ze swej natury wielowątkowe. Świadczy o tym choćby najprostszy podział na dwa wątki: wątek obsługi kodu z metodą *main*() (dla aplikacji) oraz wątek zarządzania stertą GarbageCollection (posiadający znacznie niższy priorytet). Oczywiście programista tworząc własną aplikację może zapragnąć podzielić przepływ wykonywania działań w programie na szereg własnych wątków. Java umożliwia dokonanie takiego podziału.

4.2 Tworzenie wątków w Javie

W celu napisania kodu wątku w Javie konieczne jest albo bezpośrednie dziedziczenie z klasy *Thread* lub pośrednie poprzez implementację interfejsu *Runnable*. Wykorzystanie interfejsu jest szczególnie istotne wówczas, gdy dana klasa (klasa wątku) dziedziczy już z innej klasy, a ponieważ w Javie nie może dziedziczyć z dwóch różnych klas, dlatego konieczne jest zaimplementowanie interfejsu. Każdy wątek powinien być wywołany, mieć opisane zadania do wykonania oraz posiadać zdolność do zakończenia jego działania. Funkcjonalność tą, otrzymuje się poprzez stosowanie trzech podstawowych dla wątków metod klasy *Thread*:

start() - jawnie wywołuje rozpoczęcie wątku,
run() - zawiera zestaw zadań do wykonania,
interrupt() - umożliwia przerwanie działania wątku.

Metoda *run*() nie jest wywoływana jawnie lecz pośrednio poprzez metodę *start*(). Użycie metody *start*() powoduje wykonanie działań zawartych w ciele metody *run*(). Jeśli w międzyczasie nie zostanie przerwane zadanie, w ciele którego dany wątek działa, to końcem życia wątku będzie koniec działania metody *run*(). Do innych ważnych metod opisujących działanie wątków należy zaliczyć:

static int *activeCount*():
- wywołanie tej metody powoduje zwrócenie liczby wszystkich aktywnych wątków danej grupy,

static int *enumerate*(Thread[] tarray):
- wywołanie tej metody powoduje skopiowanie wszystkich aktywnych wątków danej grupy do tablicy oraz powoduje zwrócenie liczby wszystkich skopiowanych wątków,

static void *sleep* (long ms); gdzie ms to liczba milisekund:
- wywołanie tej metody powoduje uśpienie danego wątku na czas wskazany przez liczbę milisekund;

static *yield*():
- wywołanie tej metody powoduje przerwanie wykonywania aktualnego wątku kosztem wykonywania innego wątku (jeśli taki istnieje) na Maszynie Wirtualnej. Metodę tą stosowaliśmy omawiając proces GarbageCollection, zawieszając wątek działania programu na rzecz wywołania wątku zwalniania pamięci (*System.gc*()).

Ponadto istnieją jeszcze inne metody klasy *Thread*, np. *setName*(), *setDaemon*(), *setPriority*(), *getName*(), *getPriority*(), *isDaemon*(), *isAlive*(), *isInterrupt*(), *join*(), itd., które mogą być pomocne w pracy z wątkami. Część z tych metod zostanie poniżej zaprezentowana.

W celu zobrazowania możliwości generacji wątków w Javie posłużmy się następującym przykładem:

Przykład 4.1:

```
//Los.java:

import java.util.*;

class Jasnosc extends Thread {
	Thread j;
	Jasnosc(Thread j){
		this.j=j;
	}
	public void run(){
		int n=0;
		boolean b = false;
		Random r = new Random();
		do{
			if( !(this.isInterrupted())){
				n = r.nextInt();
				System.out.println("Jasność");
			} else {
				n=200000000;
				b=true;
			}
		}while(n<200000000);
		if(b){
			System.out.println(this+" jestem przerwany, kończę pracę");
		} else {
			Thread t = Thread.currentThread();
			System.out.println("Tu wątek "+t+" Jasność");
			System.out.println("Zatrzymuję wątek: "+j);
			j.interrupt();
			System.out.println("KONIEC: Jasność");
		}
	}
}// koniec class Jasnosc

class Ciemnosc extends Thread {
	Thread c;
	public void ustawC(Thread td){
		this.c=td;
	}
	public void run(){
		int n=0;
		Random r = new Random(12345678L);
		boolean b = false;
```

```
                do{
                        if( !(this.isInterrupted())){
                                n = r.nextInt();
                                System.out.println("Ciemność ");
                        } else {
                                n=200000000;
                                b=true;
                        }
                }while(n<200000000);
                if(b){
                        System.out.println(this+" jestem przerwany, kończę pracę");
                } else {
                        if (c.isAlive()) {
                                Thread t = Thread.currentThread();
                                System.out.println("Tu wątek "+t+" Ciemność");
                                System.out.println("Zatrzymuję wątek: "+c);
                                c.interrupt();
                        } else {
                                Thread t = Thread.currentThread();
                                System.out.println("Tu wątek "+t+" jestem jedyny ");
                        }
                        System.out.println("KONIEC: Ciemność");
                }
        }
}// koniec class Ciemnosc

public class Los{

  public static void main(String args[]){
        Ciemnosc zlo = new Ciemnosc();
        Jasnosc dobro = new Jasnosc(zlo);
        zlo.ustawC(dobro);
        zlo.start();
        dobro.start();
  }

}// koniec public class Los
```

Przykładowy rezultat działania powyższego programu może być następujący:

Ciemność
Jasność
Ciemność
Jasność
Ciemność
Tu wątek Thread[Thread-1,5,main] Jasność
Ciemność
Zatrzymuję wątek: Thread[Thread-0,5,main]
Ciemność
KONIEC: Jasność
Thread[Thread-0,5,main] jestem przerwany, kończę pracę

W prezentowanym przykładzie stworzono dwie klasy dziedziczące z klasy *Thread*. Obie klasy zawierają definicję metody *run*() konieczną dla pracy danego wątku. Metody *run*() zawierają generację liczb pseudolosowych w pętli, przerywanej w momencie otrzymania wartości progowej (=> 200 000 000). Dodatkowo wprowadzono w pętli *do-while*() obu metod warunek sprawdzający czy dany wątek nie został przerwany poleceniem *interrupt*(). Jeżeli wygenerowana zostanie liczba przekraczająca wartość progową to dany wątek zgłasza kilka komunikatów oraz przerywa pracę drugiego. W programie umieszczono kontrolę działania wątków (*isAlive()* - zwraca *true*, jeżeli dany wątek wykonuje jeszcze działanie zdefiniowane w jego metodzie run()), gdyż koniec wykonywania metody run() danego wątku jest równoważne z jego eliminacją. Nie można wówczas otrzymać informacji o działającym wątku poprzez domyślną konwersję „uchwytu" obiektu metodą *toString*() w ciele instrukcji *System.out.println*(). Jeśli wątek pracuje wówczas można uzyskać o nim prostą informację zawierającą dane o jego nazwie (np. Thread-0), priorytecie (np. 5) oraz o nazwie wątku naczelnego (np. *main*): np.: *Thread*[*Thread-0,5,main*]. Efekt działania powyższego programu może być różny ponieważ wartość zmiennej sprawdzanej w warunku pętli jest generowana losowo.
Kolejny przykład ukazuje możliwość nadawania nazw poszczególnym wątkom oraz ustawiania ich priorytetów:

Przykład 4.2:

```
//Widzenie.java:
//Kod klas Ciemnosc i Jasnosc musi byś dostępny dla klasy Widzenie,
//umieszczony np. w tym samym katalogu (ten sam pakiet).

import java.util.*;

public class Widzenie{

  public static void main(String args[]){
        System.out.println("Maksymalny priorytet wątku = "+Thread.MAX_PRIORITY);
        System.out.println("Minimalny priorytet wątku = "+Thread.MIN_PRIORITY);
        System.out.println("Normalny priorytet wątku = " + Thread.NORM_PRIORITY+"\n");

        Ciemnosc zlo = new Ciemnosc();
        Jasnosc dobro = new Jasnosc(zlo);
        zlo.setName("zlo");
        zlo.setPriority(4);
        dobro.setName("dobro");
        dobro.setPriority(6);
        zlo.ustawC(dobro);
        zlo.start();
        dobro.start();
 }

}// koniec public class Widzenie
```

4.3 Priorytety

W wyniku działania pierwszych trzech instrukcji powyższego programu wyświetlone są informacje o wartościach priorytetów: maksymalnej (10), minimalnej (1) i domyślnej (5). Następnie nadano nazwy poszczególnym wątkom, tak więc nie będzie już występowała nazwa domyślna tj.: *Thread-NUMER*, lecz ta ustawiona przez programistę. Wątek o nazwie „zlo" uzyskał priorytet „4", natomiast wątek o nazwie „dobro" uzyskał priorytet „6". Oznacza to, że pierwszeństwo dostępu do zasobów komputera uzyskał wątek „dobro". W rezultacie działania programu informacje generowane przez wątek „zlo" mogą się nie pojawić na ekranie monitora. Domyślna wartość priorytetu, co było zaprezentowane w przykładzie 4.1, wynosi 5. Priorytety wątków można zmieniać w czasie wykonywania działań.

Priorytety stanowią pewien problem z punktu widzenia uniwersalności kodu w Javie. Otóż system priorytetów danej platformy (systemu operacyjnego) musi być odwzorowany na zakres 10 stanów. Przykładowo dla systemu operacyjnego Solaris liczba priorytetów wynosi 2^{31}, podczas gdy dla Windows NT aktualnych priorytetów jest praktycznie 7. Trudno jest więc ocenić czy ustawiając priorytet w Javie na 9 uzyskamy dla Windows NT priorytet 5, 6 czy może 7. Co więcej poprzez mechanizm zwany „*priority boosting*" Windows NT może zmienić priorytet wątku, który związany jest z wykonywaniem operacji wejścia/wyjścia. Oznacza to, że nie można wykorzystywać uniwersalnie priorytetów do sterowania lub inaczej do wyzwalania wątków. Co może zrobić programista aby zastosować priorytety do sterowania wątkami? Może ograniczyć się do użycia stałych *Thread.MIN_PRIORITY*, *Thread.NORM_PRIORITY* oraz *Thread.MAX_PRIORITY*.

4.4 Przetwarzanie współbieżne a równoległe

Generalnie możliwe są dwa sposoby przetwarzania wątków: współbieżnie lub równolegle. Przetwarzanie współbieżne oznacza wykonywanie kilku zadań przez procesor w tym samym czasie poprzez przeplatanie wątków (fragmentów zadań). Przetwarzanie równoległe natomiast oznacza wykonywanie kilku zadań w tym samym czasie przez odpowiednią liczbę procesorów równą ilości zadań. Teoretycznie Java umożliwia jedynie przetwarzanie współbieżne ponieważ wszystkie wątki są wykonywane w otoczeniu Maszyny Wirtualnej, będącej jednym zadaniem dla danej platformy. Można oczywiście stworzyć klika kopii Maszyny Wirtualnej dla każdego procesora, i dla nich uruchamiać wątki (które mogą się komunikować). Innym rozwiązaniem przetwarzania równoległego w Javie jest odwzorowywanie wątków Maszyny Wirtualnej na wątki danej platformy (systemu operacyjnego). Oznacza to oczywiście odejście od uniwersalności kodu.
Sterowanie wątkami (przetwarzanie wątków) związane jest więc z dwoma istniejącymi modelami: wielozadaniowość kooperacyjna, bez wywłaszczania (*cooperative multitasking*) wielozadaniowość z wywłaszczaniem (szeregowania zadań - *preemptive multitasking*). Pierwszy z nich polega na tym, że dany wątek tak długo korzysta ze swoich zasobów (procesora) aż zdecyduje się zakończyć swoją pracę. Jeżeli dany wątek kończy swoje wykonywanie to uruchamiany jest wątek (przydzielane są mu zasoby) o najwyższym priorytecie wśród tych, które czekały na uruchomienie. Taki model sterowania wątkami umożliwia jedynie współbieżność wątków a wyklucza przetwarzanie równoległe. Równoległe przetwarzanie jest możliwe jedynie wtedy, gdy obsługa wątków jest wykonana w modelu szeregowania

zadań (preemptive). W modelu tym wykorzystywany jest zegar do sterowania pracą poszczególnych wątków (do przełączania pomiędzy wątkami). Przykładem systemu operacyjnego wykorzystującego pierwszy model przełączania pomiędzy wątkami jest Windows 3.1, natomiast przykładem implementacji drugiego modelu jest Windows NT. Co ciekawe Solaris umożliwia wykorzystanie obu modeli.
Wątki są odwzorowywane na procesy danej platformy według metody zależnej od systemu operacyjnego platformy. Dla NT jeden wątek jest odwzorowywany na jeden proces (odwołanie do jądra systemu - około 600 cykli maszynowych). Dla innych systemów operacyjnych może odwzorowywanie może wyglądać inaczej. Przykładowo Solaris wprowadza pojęcie tzw. *lightweight process*, oznaczające prosty proces systemu mogący zawierać jeden lub kilka wątków. Oczywiście dany proces można przypisać do konkretnego procesora (w systemie operacyjnym). Problem zarządzania wątkami i procesami to oddzielne zagadnienie. W rozdziale tym istotny jest tylko jeden wniosek dla programisty tworzącego programy w Javie: Sposób obsługi wątków (priorytety, przełączanie, odwzorowywanie na procesy, itp.) zależy wyłącznie od Maszyny Wirtualnej, czyli pośrednio od platformy pierwotnej. Praca z wątkami nie jest więc w pełni niezależna od platformy tak, jak to zakładała pradawna idea Javy: *„write once run anywhere”*.

4.5 Przerywanie pracy wątkom

Przerwanie pracy danego wątku może być rozumiane dwojako: albo jako chwilowe wstrzymanie pracy, lub jako likwidacja wątku. Likwidacja wątku może teoretycznie odbywać się na trzy sposoby: morderstwo, samobójstwo oraz śmierć naturalna. W początkowych wersjach JDK (do 1.2) w klasie *Thread* zdefiniowana była metoda stop(), która w zależności od wywołania powodowała zabicie lub samobójstwo wątku. Niestety ponieważ w wyniku wywołania metody *stop*() powstawały częste błędy związane z wprowadzeniem bibliotek DLL w środowisku Windows w stan niestabilny, dlatego metoda *stop*() uznana została za przestarzałą (jest wycofywana - *deprecated*). Likwidacja wątku może więc odbyć się jedynie poprzez jego naturalną śmierć. Naturalna śmierć wątku wyznaczana jest poprzez zakończenie działania metody run(). Konieczne jest więc zastosowanie takiej konstrukcji metody *run*(), aby można było sterować końcem pracy wątku. Najbardziej popularne są dwie metody: pierwsza wykorzystuje wiadomość przerwania pracy wątku *interrupt*(), druga bada stan przyjętej flagi. Pierwsza metoda została wykorzystana w przykładzie 4.1. Można jednak uprościć kod programu poprzez ustawienie jednego warunku, w którym będzie wykonywana treść wątku np.:

```
while(! Thread.interrupted()) {
(...)
}
```

Druga metoda związana jest z podobnym testem flagi. Wartość flagi generowana jest w wyniku działania wyrażenia. Przykładowo można testować zgodność dwóch wątków:

```
while(watekMoj==watekObecny){
(...)
}
```

wówczas można wywołać warunek końca używając metody:

```
public void stop() {
        watekObecny = null;
}
```
, lub można testować wartość zwracaną przez funkcję:

```
while(fun(arg1,arg2));
```

gdzie
```
public boolean fun(int arg1, int arg2){

        { (...)
        return true;
        }
        {(...)
        return false;
        }
}
```

itp.

Niestety nie zawsze można zakończyć pracę wątku poprzez odpowiednie ustawienie flagi. Dlaczego, otóż dlatego, że dany wątek może być blokowany ze względu na dostęp do danych, które są chwilowo niedostępne (zablokował je inny wątek). Wówczas możliwość testowania flagi pojawia się dopiero po odblokowaniu wątku (co może bardzo długo trwać). Dlatego pewną metodą zakończenia pracy wątku jest wywołanie metody *interrupt*(), która wygeneruje wyjątek dla danego wątku. Odpowiednia obsługa wyjątku może doprowadzić do zakończenia działania metody run, a więc i wątku. Należy jednak pamiętać o tym, że generacja wyjątku może spowodować taki stan pól obiektu lub klasy, który spowoduje niewłaściwe działanie programu. Jeśli takie zjawisko może wystąpić dla tworzonej aplikacji wówczas należy ustawić odpowiednie warunki kontrolne w obsłudze wyjątku, przed zakończeniem pracy wątku. Przykładowa obsługa zakończenia pracy wątku może wyglądać następująco:

```
public void run(){
        try{
                while(! (this.isInterrupted)){
                        /*wyrażenia + instrukcja generujące wyjątek IE, np. sleep(1)*/
                }
        } catch (InterruptedException ie){
        }// instrukcja pusta-> przejście do końca metody
}
```

4.6 Przerywanie tymczasowe

Możliwe jest również tymczasowe zawieszenie pracy wątku czyli wprowadzenie go w stan *Not Runnable*. Możliwe jest to na trzy sposoby:

- wywołanie metody *sleep*();
- wywołanie metody *wait*() w celu oczekiwania na spełnienie określonego warunku;
- blokowanie wątku przez operację wejścia/wyjścia (aż do jej zakończenia).

W pierwszym przypadku stosuje się jedną z metod *sleep*() zdefiniowaną w klasie *Thread*. Metoda *sleep*() umożliwia wprowadzenie w stan wstrzymania pracy wątku na określony czas podawany jako liczba milisekund stanowiąca argument wywołania metody. W czasie zaśnięcia wątku może pojawić się wiadomość przerywająca pracę wątku (*interrupt*()) dlatego konieczna jest obsługa wyjątku *InterruptedException*. Korzystanie z metody *wait*() zdefiniowanej w klasie *Object* polega na wstrzymaniu wykonywania danego wątku aż do pojawienia się wiadomości *notify*() lub *notifyAll*() wskazującej na dokonanie zmiany, na którą czekał wątek. Te trzy metody są zdefiniowane w klasie *Object* i są dziedziczone przez wszystkie obiekty w Javie. Należy tutaj wskazać na istotną różnicę pomiędzy dwoma pierwszymi sposobami tymczasowego wstrzymywania pracy wątku. Otóż wywołanie metody *sleep*() nie powoduje utraty praw (blokady) do danych (wątek dalej blokuje monitor - o czym dalej w tym rozdziale) związanych z danym wątkiem. Oznacza to, że inny wątek, który z tych danych chce skorzystać (o ile były zablokowane) nie może tego uczynić i będzie czekał na zwolnienie blokady. To, czy dany kod (zmienne) są blokowane czy nie określa instrukcja *sychronized*, która jest opisana szerzej w dalszej części tego rozdziału. Dla odróżnienia wywołanie metody *wait*() odblokowuje dane, i czeka na kolejne przejęcie tych danych (zablokowanie) po wykonaniu pewnego działania przez inne wątki.
Rozważmy następujący przykład obrazujący możliwość zatrzymywania oraz tymczasowego wstrzymywania (*sleep*) pracy wątku.

Przykład 4.3:

```
//Walka.java:

import java.util.*;

class Imperium extends Thread {
	private String glos;

	Imperium(String s){
		this.glos=s;
	}

	public void set(String s){
		this.glos=s;
	}

	public boolean imperator(){
		String mowca = Thread.currentThread().getName();
		if (mowca.equals(glos)){
			System.out.println("Mówi Imperator !");
			return true;
		}
```

```
			return false;
		}// koniec public boolean imperator()

		public void run(){
			while(imperator());
			System.out.println("Koniec pracy Imperium");
		}

}// koniec class Imperium

class Republika extends Thread {
		private String glos;

		Republika(String s){
			this.glos=s;
		}

		public void set(String s){
			this.glos=s;
		}

		public boolean senat(){
			String mowca = Thread.currentThread().getName();
			System.out.println("Mówi Senat !");
				return true;
			}
			return false;
		}// koniec public boolean senat()

		public void run(){
			while(senat());
			System.out.println("Koniec pracy Senatu");
		}

}// koniec class Republika

class RadaJedi extends Thread {
		private String glos;
		private Imperium imp;
		private Republika rp;

		RadaJedi(String s, Imperium i, Republika r){
			this.glos=s;
			this.imp=i;
			this.rp=r;
		}

		public boolean rada(){
			String mowca = Thread.currentThread().getName();
			if (mowca.equals(glos)){
				System.out.println("Zamach stanu - Rada Jedi u władzy Senat !");
				imp.set(glos);
				rp.set(glos);
				try{
					sleep(500);
```

```
                        } catch (InterruptedException ie){
                        }
                        return false;
                }
                return true;
        }// koniec public boolean imperator()

        public void run(){
                while(rada());
                System.out.println("Koniec pracy Rady");
        }

}// koniec class RadaJedi


public class Walka{

  public static void main(String args[]){
        Imperium im = new Imperium("Imperator");
        Republika rep = new Republika("Senat");
        RadaJedi rj = new RadaJedi("Rada",im,rep);
        im.setName("Imperator");
        rep.setName("Senat");
        rj.setName("Rada");

        im.start();
        rep.start();
        try{
                Thread.currentThread().sleep(6000);
        } catch (InterruptedException ie){
        }
        rj.start();
  }

}// koniec public class Walka
```

Powyższy przykład ukazuje metodę zakończenia pracy wątku w oparciu o jego śmierć naturalną spowodowaną końcem działania metody run(). Trzy klasy wątków Imperium, Republika i RadaJedi wykorzystują ten sam mechanizm zakończenia pracy metody run() poprzez ustawienie pola glos na wartość inną niż nazwa aktualnego wątku. W metodzie głównej wprowadzono również tymczasowe wstrzymanie wykonywania wątku głównego (*Thread*[*main,5,main*]) na okres 6 sekund, celem ukazania pracy pozostałych wątków. Po okresie 6 sekund uruchomiony zostaje wątek o nazwie „Rada" powodujący zakończenie pracy wszystkich stworzonych wątków przez wątek główny.
Istnieją również inne metody tymczasowego wstrzymania pracy wątku. Pierwsza z nich powoduje chwilowe wstrzymanie aktualnie wykonywanego wątku umożliwiając innym wątkom podjęcie pracy. Metoda ta opiera się o wykorzystanie poznanej już statycznej metody *yield*(). Inne przypadki wstrzymywania pracy wątku mogą być związane z blokowaniem wątku ze względu na brak dostępu do zablokowanych danych. Wątek rozpocznie dalszą pracę dopiero wtedy, gdy potrzebne dane będą dla niego dostępne. Ciekawą sytuacją może być również wstrzymanie pracy wątku

spowodowane blokowaniem danych przez przebudzony wątek o wyższym priorytecie.
Ponieważ uruchamianie wątku jest czasochłonne dla Maszyny Wirtualnej, często stosuje się metodę zwaną jako „pull threads". Wyciąganie wątków polega na tym, że w czasie rozpoczęcia pracy programu tworzone są liczne wątki, które następnie są wprowadzane w stan czuwania i umieszczane są w pamięci (na stosie). Program w czasie pracy jeżeli potrzebuje wątku nie tworzy go, lecz pobiera z pamięci i budzi do pracy. Rozwiązanie takie jest szczególnie efektywne przy tworzeniu różnego rodzaju serwerów, kiedy czas uruchamiania programu nie jest tak istotny co czas wykonywania poleceń podczas realizacji programu. Przykładowym serwerem gdzie takie rozwiązanie może być wykorzystanie jest serwer WWW, w którym każde połączenie jest oddzielnym wątkiem.

Poniższy przykład ukazuje efekt działania metod *wait*() i *notify*() w celu tymczasowego wstrzymywania wątku i wraz ze zwolnieniem blokowanych danych w celu ich wczytania przez wątek konkurujący.

Przykład 4.4:

```
// Wojna.java:

class Strzal {
        private String strzelec = "nikt";
        private String tratata[]={ "PIF","PAF"};
        private int i=0;
        public synchronized boolean strzal(String wrog) {

                String kto = Thread.currentThread().getName();

                if (strzelec.compareTo("POKOJ") == 0)
                        return false;
                if (wrog.compareTo("POKOJ") == 0) {
                        strzelec = wrog;
                        notifyAll();
                        return false;
                }

                if (strzelec.equals("nikt")) {
                        strzelec = kto;
                         return true;
                }

                 if (kto.compareTo(strzelec) == 0) {
                         System.out.println(tratata[i]+"! ("+strzelec+")");
                         strzelec = wrog;
                         i=1-i;
                         notifyAll();
                 } else {
                 try {
                        long zwloka = System.currentTimeMillis();
                        wait(200);
                        if ((System.currentTimeMillis() - zwloka) > 200) {
                         System.out.println("Tu "+kto+", czekam na ruch osobnika:"+ strzelec);
```

```
                }
            } catch (InterruptedException ie) {
            }
          }
          return true;
    }
}//koniec class Strzal


class Strzelec implements Runnable {
    Strzal s;
    String wrogPubliczny;

    public Strzelec(String wrog, Strzal st) {
        s = st;
        wrogPubliczny = wrog;
    }

    public void run() {
        while (s.strzal(wrogPubliczny)) ;
    }
}// koniec class Strzelec


public class Wojna {
    public static void main(String args[]) {
        Strzal st = new Strzal();
        Thread luke = new Thread(new Strzelec("Vader", st));
        Thread vader = new Thread(new Strzelec("Luke",st));
        luke.setName("Luke");
        vader.setName("Vader");
        luke.start();
        vader.start();

        try {
            Thread.currentThread().sleep(1000);
        } catch (InterruptedException ie) {
        }

        st.strzal("POKOJ");
        System.out.println("Nastał pokój!!!");
    }
}// koniec public class Wojna
```

Przykładowy program skonstruowano z trzech klas. Pierwsza z nich jest odpowiedzialna za wysyłanie komunikatów w zależności od podanej nazwy zmiennej. Jeżeli nazwa strzelca jest inna niż bieżąca (czyli inna niż tego do kogo należy ruch) to następuje oczekiwanie na nowego strzelca. Druga klasa tworzy definicję metody run() konieczną dla opisania działania wątków. Ostatnia główna klasa programu tworzy obiekt klasy opisującej proces strzelania (klasy pierwszej) a następnie uruchamia dwa wątki podając nazwy wrogich strzelców względem danego wątku. Strzelanie trwa jedną sekundę po czym następuje „POKOJ”, czyli przerwanie pracy wątków poprzez zakończenie pracy metod run() (w pętli *while*() pojawia się wartość logiczna *false*). Efektem działania programu jest następujący wydruk:

```
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
Nastał pokój!!!
```

Warto wspomnieć coś na temat stanów wątku. Do podstawowych należy zaliczyć te opisywane przez metody *isAlive*() oraz *isInterrupted*(). Metoda *isAlive*() zwraca wartość logiczną *true* jeżeli dany wątek został uruchomiony (start()) i nie został jeszcze zakończony (nie umarł). Metoda *isAlive*() nie dostarcza informacji o tym, czy wykonywanie danego wątku jest zawieszone tymczasowo czy nie (brak rozróżnienia pomiędzy wątkiem *Runnable* i *Not Runnable*). Metoda *isInterrupted*() zwraca wartość *true* jeżeli dla danego wątku została przesłana wiadomość (wykonano metodę) *interrupt*().

4.7 Synchronizowanie wątków

Jeśli dwa wątki konkurują w celu uzyskania praw dostępu do danego obiektu to zachowanie kodu zależeć może od tego, który wątek wygra. Zjawisko *te (race condition* - wyścig) jest najczęściej bardzo niekorzystne ze względu na brak kontroli (nieokreśloność) w zapewnianiu kolejności obsługi wiadomości generowanych przez różne wątki do tego samego obiektu.
Załóżmy, że istnieje następująca klasa:

```
class Test{
	int n=0;
```

```
        int m=1;
        void zamianaLP(){
                n=m;
        }
        void zamianaPL(){
                m=n;
        }
}// koniec class Test
```

oraz stworzone są dwa wątki, jeden wykonuje metodę zamianaLP(), drugi wykonuje metodę zamianaPL(). O ile oba wątki są równouprawnione i wykonywane są równolegle, to problem polega na określeniu kolejności działania, czyli określeniu jakie wartości końcowe przyjmą pola n i m. obiektu klasy Test. Możliwe są tu różne sytuacje. Każda z dwóch metod wykonuje dla swoich potrzeb aktualne kopie robocze zmiennych. Następnie po wykonaniu zadania wartości tych zmiennych są przypisywane do zmiennych oryginalnych przechowywanych w pamięci głównej. Jak łatwo się domyśleć możliwe są więc następujące stany końcowe zmiennych:
- n=0,m=0; wartość zmiennej n została przepisana do m;
- n=1, m=1; wartość zmiennej m została przepisana do n;
- n=1, m=0; wartości zmiennych zostały zamienione.

W większości przypadków (poza generatorem pseudolosowym) zjawisko wyścigu jest niekorzystne. Konieczne jest więc zastosowanie takiej konstrukcji kodu aby można było to zjawisko wyeliminować. Zanim jednak zaprezentowane zostanie rozwiązanie problemu warto przeanalizować konkretny przykład.

Przykład 4.5:

```
//WyscigiNN.java:


public class WyscigiNN implements Runnable{
        private int n;

        WyscigiNN(int nr){
                this.n=nr;
        }
        void wyswietl(int i){
                System.out.println("Dobro = "+i);
                System.out.println("Zlo = "+i);
        }
        void zmianaZla(){
                System.out.print("ZLO: ");
                for(int l=0; l<10;l++,++n){
                        System.out.print(Thread.currentThread().getName()+": "+n+" ");
                }
                System.out.println(" ");
        }
        void zmianaDobra(){
                System.out.print("DOBRO: ");
                for(int l=0; l<20;l++,--n){
                        System.out.print(Thread.currentThread().getName()+": "+n+" ");
```

```
            }
            System.out.println(" ");
    }

    public void run(){
                    while (!(Thread.interrupted())){
                            if ( (Thread.currentThread().getName()).equals("T1")){
                                    zmianaZla();
                            } else
                                    zmianaDobra();
                    }
    }

    public static void main(String args[]){
            WyscigiNN w1= new WyscigiNN(100);
            Thread t1,t2;
            t1= new Thread(w1);
            t2=new Thread(w1);
            t1.setName("T1");
            t2.setName("T2");
            t2.start();
            t1.start();

            try{
                    Thread.currentThread().sleep(600);
            } catch (InterruptedException ie){
            }
            t1.interrupt();
            t2.interrupt();
            try{
                    Thread.currentThread().sleep(2000);
            } catch (InterruptedException ie){
            }
            System.out.println("\n");
            w1.wyswietl(w1.n);
    }
}// koniec public class WyscigiNN
```

Powyższy program umożliwia obserwację zjawiska wyścigu. Klasa główna aplikacji implementuje interfejs *Runnable* w celu definicji metody run() koniecznej dla pracy wątków. W ciele klasy głównej zadeklarowano jedną zmienną prywatną typu *int* oraz pięć metod. Pierwsza metod wyswietl() umożliwia wydruk wartości pola obiektu (zmiennej n). Druga i trzecia metoda, a więc zmianaZla() i zmianaDobra() wykonują wydruk modyfikowanej wartości pola obiektu (n). Każda drukowana wartość poprzedzana jest nazwa wątku („T1” lub T2”), który wywołał daną metodę. W metodzie *run*() w pętli warunkowej określającej koniec działania wątku (przerwanie - *interrupt*()) wywoływana jest albo metoda zmianaZla() albo zmianaDobra() w zależności od nazwy aktualnego wątku. Metoda statyczna aplikacji zawiera inicjację obiektu klasy głównej, inicjację obiektów wątków (inicjacja nie jest równoważna z rozpoczęciem pracy wątku; wątek nie jest obiektem), nadanie im nazw i ich uruchomienie. W celu obserwacji uruchomionych wątków wstrzymuje się wykonywanie wątku głównego (*main*) na okres 600 milisekund (*sleep*(600)). Następnie wysyłane są wiadomości przerwania pracy wątków, co powoduje

zakończenie działania metod run(). Kolejne uśpienie wątku głównego ma na celu wypracowanie czasu na zakończenie pracy przerywanych wątków zanim wyświetlone zostaną ostateczne rezultaty, czyli wartość pola obiektu (n). W wyniku działania programu można uzyskać następujący wydruk:

ZLO: DOBRO: T1: 100 T2: 100 T1: 101 T2: 100 T1: 101 T2: 100 T1: 100 T1: 101 T1: 102 T1: 103 T1: 104 T1: 105 T1: 106 T2: 107 ZLO: T1: 106 T1: 107 T1: 108 T1: 109 T1: 110 T1: 111 T2: 106 T1: 112 T2: 111 T1: 112 T2: 111 T1: 112 T2: 111 T1: 112 T2: 111
T2: 111 T2: 110 ZLO: T2: 109 T2: 108 T2: 107 T2: 106 T2: 105 T2: 104 T2: 103 T2: 102 T2: 101
DOBRO: T2: 100 T2: 99 T2: 98 T2: 97 T2: 96 T2: 95 T2: 94 T2: 93 T2: 92 T2: 91 T2: 90 T2: 89 T2: 88 T2: 87 T2: 86 T2: 85 T2: 84 T2: 83 T2: 82 T2: 81
DOBRO: T2: 80 T2: 79 T2: 78 T2: 77 T2: 76 T2: 75 T2: 74 T1: 73 T1: 74 T1: 75 T2: 75 T1: 76 T2: 75 T1: 76 T1: 76 T1: 77 T1: 78 T1: 79 T1: 80
ZLO: T1: 81 T1: 82 T1: 83 T1: 84 T1: 85 T1: 86 T1: 87 T1: 88 T1: 89 T1: 90
ZLO: T1: 91 T1: 92 T1: 93 T1: 94 T1: 95 T1: 96 T1: 97 T2: 75 T2: 96 T2: 95 T2: 94 T2: 93 T2: 92 T2: 91 T2: 90 T2: 89 T2: 88 T2: 87
DOBRO: T2: 86 T2: 85 T2: 84 T2: 83 T2: 82 T2: 81 T2: 80 T2: 79 T2: 78 T2: 77 T2: 76 T2: 75 T2: 74 T2: 73 T2: 72 T2: 71 T2: 70 T2: 69 T2: 68 T2: 67
DOBRO: T2: 66 T2: 65 T2: 64 T2: 63 T2: 62 T2: 61 T2: 60 T2: 59 T2: 58 T2: 57 T2: 56 T1: 57 T2: 56 T1: 57 T2: 56 T1: 57 T2: 56
T2: 56 T2: 55 T2: 54 T2: 53 T2: 52 T2: 51

Dobro = 50
Zlo = 50

Łatwo zauważyć, że otrzymano nieregularne przeplatanie wątków T1 i T2, powodujące różne operację na polu obiektu. Ponieważ T1 i T2 są równomiernie uprawnione do dostępu do pola, uzyskano różne wartości (nieregularne - co uwypuklono na wydruku podkreśleniami tekstu) tego pola w czasie działania nawet tej samej pętli for metody zmianaZla() lub zamianaDobra(). Oznacza to sytuację, gdy jeden wątek korzysta z pola, które jest właśnie zmienione przez inny wątek. Ta sytuacja jest właśnie określana mianem „*race condition*".

Konieczny jest więc tutaj mechanizm zabezpieczenia danych, w ten sposób, że jeżeli jeden wątek korzysta z nich to inne nie mogą z nich korzystać. Najprostszym rozwiązaniem byłoby zablokowanie fragmentu kodu, z którego korzysta dany wątek tak, że inne wątki muszą czekać tak długo, aż wątek odblokuje kod. Chroniony region kodu jest często nazywany monitorem (obowiązujące pojęcie w Javie). Monitor jest chroniony poprzez wzajemnie wykluczające się semafory. Oznacza to, że stworzenie monitora oraz ustawienie jego semafora (zamknięcie monitora) powoduje sytuację taką, że żaden wątek nie może z tego monitora korzystać. Jeżeli zmieni się stan semafora (zwolnienie danych) wówczas inny wątek może ten monitor sobie przywiązać (ustawić odpowiednio semafor tego wątku). W Javie blokada monitora odbywa się poprzez uruchomienie metody lub kodu oznaczonej jako *synchronized*. Jeżeli występuje dla danego wątku kod oznaczony jako *synchronized*, to kod ten staje się kodem chronionym i jego uruchomienie jest równoważne z ustanowieniem blokady na tym kodzie (o ile monitor ten nie jest już blokowany). Słowo kluczowe *synchronized* stosuje się jako oznaczenie metody (specyfikator) lub jako instrukcja. Przykładowo jeżeli metoda zmiana() przynależy do danego obiektu wówczas zapis:

```
synchronized void zmiana(){
        /*wyrażenia*/
}
```

jest równoważny praktycznie zapisowi:

```
void zmiana(){
        synchronized (this){
                /*wyrażenia*/
        }
}
```

Pierwszy zapis oznacza metodę synchronizowaną a drugi instrukcję synchronizującą. Blokada zakładana dla kodu oznaczonego poprzez *synchronized*, powoduje najpierw obliczenie odwołania (uchwytu) do danego obiektu (*this*) i założenie blokady. Wówczas żaden inny wątek nie będzie miał dostępu do monitora danego obiektu. Inaczej ujmując, żaden inny wątek nie może wykonać metody oznaczonej jako synchronized. Jeżeli zakończone zostanie działanie tak oznaczonej metody wówczas blokada jest zwalniana. Niestety stosowanie specyfikatora *synchronized* powoduje zwolnienie pracy metody nieraz i dziesięciokrotnie. Dlatego warto stosować instrukcję *synchronized* obejmując blokiem tylko to, co jest niezbędne.

W Javie (podobnie jak w innych językach programowania współbieżnego) możliwe są dwa typy obszarów działania wątku: dane dynamiczne - obiekt oraz dane statyczne. Obiekt, a więc konkretne i jednostkowe wystąpienie danej klasy jest opisywany poprzez pola i metody (w programowaniu obiektowym często nazywane z j. angielskiego jako: *instance variables* {*fields*, *methods*}). Klasa może być również opisywana poprzez pola i metody niezmienne (statyczne - *static*) dla dowolnego obiektu tej klasy (w języku angielskim elementy te nazywane są czasem *class variables{fields, methods*}). Pola i metody statyczne opisują więc stan wszystkich obiektów danej klasy, podczas gdy pola i metody obiektu opisują stan danego obiektu. Dualizm ten ma również swoje następstwa w sposobie korzystania z omawianych elementów przez wątki. Omawiany do tej pory monitor jest związany z obiektem danej klasy. Fragment kodu oznaczony przez słowo kluczowe *synchronized* (np. metoda) może być wykonywana równocześnie przez różne wątki, lecz pod warunkiem, że związane z nimi obiekty otrzymujące dane wiadomości są różne. Oznacza to, że w czasie wykonywania metody oznaczonej jako *synchronized* blokowany jest monitor danego obiektu, tak że inne wątki nie mają do niego dostępu. Poza monitorem spotyka się w literaturze [1][2][3] pojęcie kodu krytycznego - *critical section*. W [3] kod krytyczny jest definiowany jako ten, w którym wykonywany jest dostęp do tego samego obiektu z kilku różnych wątków. Kod taki, bez względu na to czy dotyczy klasy (*static*) czy obiektu oznaczany może być poprzez blok ze słowem kluczowym *synchronized*. W [2] ukazano jednak ścisły rozdział pomiędzy pojęciem *„critical section*” a monitorem: kod krytyczny to ten związany z klasą (a więc z kodem statycznym), monitor natomiast jest związany z obiektem danej klasy. Oznacza to, że kod krytyczny to taki kod, który może być wykonywany tylko poprzez jeden wątek w czasie! Nie możliwe jest bowiem stworzenie wielu obiektów z jednostkowymi metodami typu *static* (każdy obiekt widzi to samo pole lub metodę typu *static*). Oczywiście druga interpretacja jest słuszna. Dlaczego? Otóż jak już wiadomo metody i pola statyczne danej klasy należą do obiektu klasy *Class* związanego z daną metodą. Obiekt ten posiada również swój monitor, gdzie blokowanie następuje poprzez wywołanie instrukcji *synchronized static*. Maszyna Wirtualna Javy implementuje blokowanie monitora poprzez swoją instrukcję *monitorenter*, natomiast

zwalnia blokadę poprzez wywołanie instrukcji *monitorexit*. Niestety monitor obiektu klasy *Class* nie jest związany z monitorami obiektów jednostkowych danej klasy. Oznacza to, że metoda obiektu oznaczona jako *synchronized* (blokowany jest więc monitor - dostęp do obiektu) ma dostęp do pól statycznych (pola te nie należą bowiem do obiektu, czyli nie są blokowane). Rozpatrzmy następujący fragment kodu:

Przykład 4.6:

```
//KolorMiecza.java

class KolorMiecza{
	private static Color k = Color.red

	synchronized public ustawKolor(Color kolor){
		this.k=kolor;
	}
}// koniec class KolorMiecza
```

Jeżeli dwa wątki wywołają równocześnie metody ustawKolor() dla dwóch różnych obiektów klasy KolorMiecza to nie wiadomo jaką wartość będzie przechowywało pole klasy k. Występuje więc tutaj wyścig „*race condition*” czyli dwa wątki modyfikują (rywalizują o) tę samą zmienną. Oznacza to, że zablokowanie obiektu klasy KolorMiecza nie oznacza zablokowania odpowiadającemu mu obiektu klasy *Class*, czyli pola statyczne nie są wówczas blokowane. Dlatego bardzo ważne jest rozróżnienie pomiędzy monitorem (kodem obiektu) a kodem krytycznym (kodem klasy).

Na zakończenie omawiania problemu wykorzystywania instrukcji *synchronized* warto zmodyfikować prezentowany wcześniej przykład WyscigiNN.java dodając do metod zmianaZla() oraz zmianaDobra() instrukcje lub specyfikator *synchronized*.

Przykład 4.7:

```
// WyscigiSN.java:

public class WyscigiSN implements Runnable{
	private int n;

	WyscigiSN(int nr){
		this.n=nr;
	}
	 void wyswietl(int i){
		System.out.println("Dobro = "+i);
		System.out.println("Zlo = "+i);
	}
	 void zmianaZla(){
		synchronized (this){
		 System.out.print("ZLO: ");
		 for(int l=0; l<10;l++,++n){
			System.out.print(Thread.currentThread().getName()+": "+n+" ");
		 }
		 System.out.println(" ");
		}
```

```
 }
 void zmianaDobra(){
            synchronized (this){
             System.out.print("DOBRO: ");
             for(int I=0; I<20;I++,--n){
                        System.out.print(Thread.currentThread().getName()+": "+n+" ");
             }
             System.out.println(" ");
            }
}

public void run(){
                        while (!(Thread.interrupted())){
                                    if ( (Thread.currentThread().getName()).equals("T1")){
                                                zmianaZla();
                                    } else
                                                zmianaDobra();
                        }
}

public static void main(String args[]){
            WyscigiSN w1= new WyscigiSN(100);
            Thread t1,t2;
            t1= new Thread(w1);
            t2=new Thread(w1);
            t1.setName("T1");
            t2.setName("T2");
            t2.start();
            t1.start();

            try{
                        Thread.currentThread().sleep(600);
            } catch (InterruptedException ie){
            }
            t1.interrupt();
            t2.interrupt();
            try{
                        Thread.currentThread().sleep(2000);
            } catch (InterruptedException ie){
            }
            System.out.println("\n");
            w1.wyswietl(w1.n);
}
}// koniec public class WyscigiSN
```

Rezultat działanie powyższego programu może być następujący:

ZLO: T1: 100 T1: 101 T1: 102 T1: 103 T1: 104 T1: 105 T1: 106 T1: 107 T1: 108 T1: 109
DOBRO: T2: 110 T2: 109 T2: 108 T2: 107 T2: 106 T2: 105 T2: 104 T2: 103 T2: 102 T2: 101 T2: 100 T2: 99 T2: 98 T2: 97 T2: 96 T2: 95 T2: 94 T2: 93 T2: 92 T2: 91
ZLO: T1: 90 T1: 91 T1: 92 T1: 93 T1: 94 T1: 95 T1: 96 T1: 97 T1: 98 T1: 99
DOBRO: T2: 100 T2: 99 T2: 98 T2: 97 T2: 96 T2: 95 T2: 94 T2: 93 T2: 92 T2: 91 T2: 90 T2: 89 T2: 88 T2: 87 T2: 86 T2: 85 T2: 84 T2: 83 T2: 82 T2: 81
ZLO: T1: 80 T1: 81 T1: 82 T1: 83 T1: 84 T1: 85 T1: 86 T1: 87 T1: 88 T1: 89
DOBRO: T2: 90 T2: 89 T2: 88 T2: 87 T2: 86 T2: 85 T2: 84 T2: 83 T2: 82 T2: 81 T2: 80 T2: 79 T2: 78 T2: 77 T2: 76 T2: 75 T2: 74 T2: 73 T2: 72 T2: 71

ZLO: T1: 70 T1: 71 T1: 72 T1: 73 T1: 74 T1: 75 T1: 76 T1: 77 T1: 78 T1: 79
DOBRO: T2: 80 T2: 79 T2: 78 T2: 77 T2: 76 T2: 75 T2: 74 T2: 73 T2: 72 T2: 71 T2: 70 T2: 69 T2: 68 T2: 67 T2: 66 T2: 65 T2: 64 T2: 63 T2: 62 T2: 61
ZLO: T1: 60 T1: 61 T1: 62 T1: 63 T1: 64 T1: 65 T1: 66 T1: 67 T1: 68 T1: 69
DOBRO: T2: 70 T2: 69 T2: 68 T2: 67 T2: 66 T2: 65 T2: 64 T2: 63 T2: 62 T2: 61 T2: 60 T2: 59 T2: 58 T2: 57 T2: 56 T2: 55 T2: 54 T2: 53 T2: 52 T2: 51
ZLO: T1: 50 T1: 51 T1: 52 T1: 53 T1: 54 T1: 55 T1: 56 T1: 57 T1: 58 T1: 59

Dobro = 60
Zlo = 60

Rezultat jasno przedstawia, że poszczególne metody są wykonywane sekwencyjnie, co oznacza blokowanie dostępu do metody (pola) przez wątek, który z niej korzysta. W wydruku wyraźnie można zaobserwować kolejne realizacje pętli for ujętych w metodach zmianaZla() i zmianaDobra().
Należy na koniec dodać, że możliwa jest sytuacja taka, że wszystkie istniejące wątki będą się wzajemnie blokować. Wówczas żaden kod nie jest wykonywany i powstaje tzw. impas, zakleszczenie (*deadlock*). Java nie dostarcza specjalnych mechanizmów detekcji impasu. Programista musi niestety przewidzieć ewentualną możliwość wystąpienia totalnej blokady, i temu zaradzić modyfikując kod.

4. 8 Grupy wątków – ThreadGroup

Poszczególne wątki można z sobą powiązać poprzez tworzenie grup wątków. Grupy wątków są przydatne ze względu na sposób organizacji pracy programu, a co za tym idzie możliwością sterowania prawami wątków. Przykładowo inne powinny być uprawnienia zawiązane z wątkami apletu niż te, związane z aplikacjami. Grupowanie wątków możliwe jest dzięki zastosowaniu klasy *ThreadGroup*. Stworzenie obiektu tej klasy umożliwia zgrupowanie nowo tworzonych wątków poprzez odwołanie się do obiektu *ThreadGroup* w konstruktorze każdego z wątków, np. *Thread(ThreadGroup tg, String nazwa).* Ponieważ w Javie wszystkie obiekty mają jedno główne źródło, również i dla obiektów typu *ThreadGroup* można przeprowadzić analizę hierarchii obiektów. Dodatkowo każdy obiekt klasy *ThreadGroup* może być jawnie stworzony jako potomek określonej grupy wątków, np. poprzez wywołanie konstruktora *ThreadGroup(ThreadGroup rodzic, String nazwa).* Maszyna Wirtualna Javy pracuje więc ze zbiorem zorganizowanymi w grupy wątków. Poniższa aplikacja ukazuje wszystkie pracujące wątki na danej platformie Javy:

Przykład 4.8:

```
//Duchy.java

public class Duchy {

  public static void main(String args[]) {

        ThreadGroup grupa = Thread.currentThread().getThreadGroup();
        while(grupa.getParent() != null) {
                grupa = grupa.getParent();
        }
        Thread[] watki = new Thread[grupa.activeCount()];
```

```
            grupa.enumerate(watki);
            for (int k = 0; k < watki.length; k++) {
                        System.out.println(watki[k]);
            }
  }
}// koniec public class Duchy
```

W powyższym przykładzie zastosowano pętlę *while*, w ciele której poszukiwany jest obiekt klasy *ThreadGroup* będący na korzeniem w drzewie wszystkich grup wątków. Następnie dla tak określonego obiektu tworzona jest tablica obiektów klasy *Thread* o rozmiarze wynikającym z liczby aktywnych wątków zwracanych metodą *activeCount*(). W kolejnym kroku za pomocą metody *enumerate*() inicjowana jest tablica obiektami wątków podstawowej grupy wątków. Prezentacja otrzymanych wyników wykonana jest poprzez znaną już konwersję wątków na tekst. W rezultacie można uzyskać następujący wynik:

Thread[Signal dispatcher,5,system]
Thread[Reference Handler,10,system]
Thread[Finalizer,8,system]
Thread[main,5,main]

Metody klasy *ThreadGroup*, oprócz tych już poznanych (*activeCount*(), *enumerate*(), *getParent*()) są podobne do tych zdefiniowanych dla klasy Thread, lecz dotyczą obiektów klasy *ThreadGroup*, np. *getName*() – zwraca nazwę grupy; *interrupt*() – przerywa wszystkie wątki w grupie, itp. Ciekawą metodą jest metoda list() wysyłająca na standardowe urządzenie wyjścia (np. ekran) informacje o danej grupie. Zastosowanie tej metody w powyższym przykładzie znacznie skraca kod źródłowy:

Przykład 4.9:

```
//Duchy1.java

public class Duchy1 {

  public static void main(String args[]) {

            ThreadGroup grupa = Thread.currentThread().getThreadGroup();
            while(grupa.getParent() != null) {
                        grupa = grupa.getParent();
            }
            grupa.list();
  }
}// koniec public class Duchy1
```

W rezultacie działania tej metody można uzyskać następujący wydruk na ekranie:

```
java.lang.ThreadGroup[name=system,maxpri=10]
   Thread[Signal dispatcher,5,system]
   Thread[Reference Handler,10,system]
   Thread[Finalizer,8,system]
   java.lang.ThreadGroup[name=main,maxpri=10]
      Thread[main,5,main]
```

Z wydruku tego jasno widać, że na danej platformie Javy aktywne są dwie grupy wątków: główna o nazwie „system" oraz potomna o nazwie „*main*". W grupie głównej znajduje się między innymi wątek o nazwie „*Finalizer*", związany z wykonywaniem zadań w czasie niszczenia obiektów. Jak wspomniano wcześniej w tym materiale zwalnianiem pamięci w *Javie* zajmuje się wątek *GarbageCollection*. Programista praktycznie nie ma na niego wpływu. Problem jednak polega na tym, że czasami konieczne jest wykonanie pewnych działań w czasie niszczenia obiektu, innych niż zwalnianie pamięci (np. zamknięcie strumienia do pliku). Obsługą takich zleceń zajmuje się wątek *Finalizer* (wykonywanie kodu zawartego w metodach *finalize*()).
Ciekawostką może być inny sposób uzyskania raportu o działających wątkach w systemie. Otóż w czasie działania programu w Javie należy w konsoli przywołać instrukcję przerwania: dla Windows – *CTRL-BREAK*, dla Solaris *kill –QUIT* (dla procesu Javy). Po instrukcji przerwania następuje wydruk stanu wątków i oczywiście przerwanie programu Javy.

4.9 Demony

Wszystkie wątki stworzone przez użytkownika giną w momencie zakończenia wątku naczelnego (zadania). Jeżeli programista pragnie stworzyć z danego wątku niezależny proces działający w tle (demon) wówczas musi dla obiektu danego wątku podać jawnie polecenie deklaracji demona: *setDaemon(true*). Metoda ta musi być wywołana zanim rozpoczęte zostanie działanie wątku poprzez zastosowanie metody start(). Poprzez odwołanie się do „uchwytu" wątku można sprawdzić czy jest on demonem czy nie. Do tego celu służy metoda *isDaemon()* zwracająca wartość *true* jeżeli dany wątek jest demonem. Rozważmy następujący przykład:

Przykład 4.10:

```
//Demony.java

public class Demony extends Thread{

  public void run(){
		while(!Thread.interrupted()){
		}
}
  public static void main(String args[]) {
		Demony d = new Demony();
		d.setName("DEMON");
		d.setDaemon(true);
		d.start();
		ThreadGroup grupa = Thread.currentThread().getThreadGroup();
		while(grupa.getParent() != null) {
			grupa = grupa.getParent();
		}
		Thread[] watki = new Thread[grupa.activeCount()];
		grupa.enumerate(watki);
		for (int k = 0; k < watki.length; k++) {
			if (watki[k].isDaemon()) System.out.println("Demon: "+watki[k]);
		}
```

```
        d.interrupt();
  }
}// koniec public class Demony
```

Powyższy program jest przerobioną wersją aplikacji Duchy.java tworzącą nowego demona w systemie oraz drukującą tylko te wątki, które są demonami. Wynik działania tego programu jest następujący:

Demon: Thread[Signal dispatcher,5,system]
Demon: Thread[Reference Handler,10,system]
Demon: Thread[Finalizer,8,system]
Demon: Thread[DEMON,5,main]

Okazuje się więc, że jedynym wątkiem w Javie po uruchomieniu programu (czyli bez własnych wątków) nie będącym demonem jest wątek *main*. Demon jest więc wątkiem działającym w tle. Jeżeli wszystkie istniejące wątki są demonami to Maszyna Wirtualna kończy pracę. Podobnie można oznaczyć grupę wątków jako demon. Nie oznacza to jednak, że automatycznie wszystkie wątki należące do tej grupy będą demonami.
Poniższy przykład ukazuje sytuację, w której wszystkie aktualnie wykonywane wątki przez Maszynę Wirtualną stanją się demonami. Wówczas maszyna kończy pracę.

Przykład 4.11:

```
//Duch.java

class Zly extends Thread{
        Zly(){
                super();
                setName("Zly_demon");
                //setDaemon(true);
        }
        public void run(){
                while(!this.isInterrupted()){

                }
        }
}// koniec class Zly

class Cacper extends Thread{

        Cacper(ThreadGroup g, String s){
                super(g,s);
        }

        public void run(){

                Zly z = new Zly();
                z.start();

                while(!this.isInterrupted()){

                }
```

```
        }

}//koniec class Cacper

public class Duch {

  public static void main(String args[]) {

        ThreadGroup zle_duchy = new ThreadGroup("zle_duchy");
        Cacper c = new Cacper(zle_duchy,"cacper");
        c.start();
        try{
                Thread.currentThread().sleep(4000);
        }catch (Exception e){}
        ThreadGroup grupa = Thread.currentThread().getThreadGroup();
        while(grupa.getParent() != null) {
                grupa = grupa.getParent();
        }
        grupa.list();
        c.interrupt();
        try{
                Thread.currentThread().sleep(4000);
        }catch (Exception e){}

        grupa = Thread.currentThread().getThreadGroup();
        while(grupa.getParent() != null) {
                grupa = grupa.getParent();
        }
        grupa.list();
  }
}// koniec public class Duch
```

W powyższym kodzie stworzono trzy wątki: „main”-> „cacper” -> „Zly_demon”. Po określonym czasie (4 sekundy) wątek „cacper” ginie śmiercią naturalną, zostawiając pracujący wątek „Zly_demon”. Po uruchomieniu tego programu pokażą się dwa wydruki o stanie wątków, po czym program zawiesi się (wieczna pętla wątku „Zly_demon”). Co ciekawe, jeżeli wywoła się polecenie wydruku stanu wątków (CTRL-Break dla Windows) okaże się, że brak jest wątku o nazwie „main”. Jeżeli powyższy kod ulegnie zmianie w ten sposób, że wątek „Zly_demon” uzyska status demona, wówczas Maszyna Wirtualna przerwie pracę nawet jeżeli wątek „Zly_demon” wykonuje wieczne działania.

Bibliografia

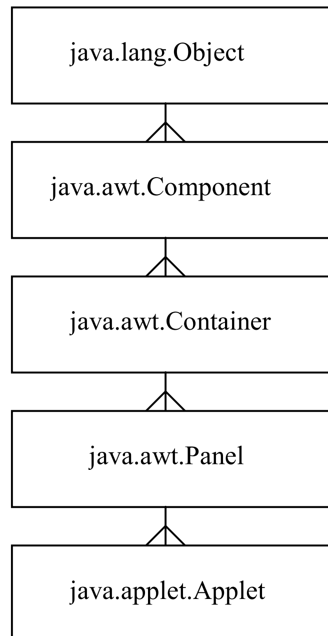
[1] Sun, Java Language Specification, Sun 1998
[2] Allen Hollub, Java Toolbox: Programming Java threads in the real world, JavaWorld, http://www.javaworld.com/jw-04-1999/jw-04-toolbox.html, 1999
[3] Sun, The Java Tutorial, Sun 1998.

Rozdział 5 Aplety, grafika w Javie

5.1 Aplety
5.2 Grafika w Javie
5.2.1 Komponenty
5.2.2 Kontenery
5.2.3 Rozkłady
5.2.4 Zdarzenia

5.1 Aplety

Aplet jest programem komputerowym, stworzonym w ten sposób, że możliwe jest jego wykonywania tylko z poziomu innej aplikacji. Oznacza to, że aplet nie jest samodzielnym programem. Jak wspomniano już na początku tego materiału, definicja apletu odbywa się poprzez definicję klasy dziedziczącej z klasy Aplet. Konieczne jest więc jawne importowanie pakietu *java.applet.** zawierającego tylko jedną klasę *Applet*. Klasa *Applet* dziedziczy w następujący sposób z klasy *Object*:



Rysunek 5.1 Dziedziczenie dla klasy Applet.

Nadklasą klasy *Applet* jest klasa Panel zdefiniowana w graficznym *pakiecie AWT – Abstract Window Toolkit*. Łatwo się więc domyśleć, że również i aplet ma formę graficzną. Można w dużym uproszczeniu powiedzieć, że pole robocze apletu jest oknem graficznym, w którym wszystkie operacje edycyjne wykonuje się przez odwołanie do obiektu graficznego klasy *Graphics*. AWT zostanie omówiona poniżej i tam również zaprezentowane zostaną liczne przykłady apletów związane z grafiką.

Jak już wspomniano aplet jest programem wykonywany pod kontrolę innej aplikacji. Przykładową aplikacją może być przeglądarka stron HTML (np. Netscae Navigator, MS Internet Explorer, itp.). Oznacza to, że konieczne jest umieszczenie wywołania kodu apletu w ciele strony HTML obsługiwanej przez daną przeglądarkę. W tym celu

wprowadzono w HTML następujące tagi: <APPLET> oraz </APPLET> oznaczające odpowiednio początek i koniec pól definiujących aplet. Podstawowym polem (atrybutem) jest pole CODE informujące aplikację czytającą HTML gdzie znajduje się skompilowany kod apletu. Przykładowo w omawianym już przykładzie 1.4 atrybut CODE zdefiniowano następująco: *code=Jedi2.class.* W przykładzie tym aplikacja ma pobrać kod apletu umieszczony w pliku o nazwie Jedi2.class znajdujący się w aktualnym katalogu. W celu wskazania ewentualnego katalogu, w którym umieszczono kod apletu można posłużyć się polem CODEBASE, np. *code=Jedi2.class codebase=klasy*. Aplet może posiadać swoją nazwę określoną w kodzie HTML poprzez wykorzystanie pola NAME, np. *code=Jedi2.class name=aplecik.* Inne bardzo popularne pola wykorzystywane do opisu apletu to WIDTH oraz HEIGHT. Oba parametry oznaczają rozmiar (szerokość i wysokość) pola pracy apletu zawartego na stronie WWW. Przykładowo *code=Jedi2.class width=200 height=100* oznacza ustalenie pola pracy apletu na stronie WWW o szerokości 200 pikseli i wysokości 100 pikseli. Możliwe jest również sterowanie położeniem okna względem tekstu poprzez przypisanie odpowiedniej wartości do pola ALIGN, takich jak: LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM oraz ABSBOTTOM np. align=middle. Czasami przydatne jest odseparowanie pola apletu od otaczającego tekstu. Wykonuje się to poprzez właściwe ustawienie liczby pikseli stanowiących spację w poziomie i pionie, czyli poprzez wykorzystanie pól: HSPACE i VSPACE. Ponieważ z różnych przyczyn nie zawsze przeglądarka będzie w stanie uruchomić aplet dlatego warto wprowadzić zastępczy tekst co wykonuje się poprzez przypisanie tekstu do pola ALT, np. alt="Aplet rysujący znak firmowy"

W pracy z apletami istotny jest również fakt, że często koniecznie trzeba wykorzystać kilka różnych plików związanych z apletem (np. kilka klas, obrazki, dźwięki, itp.). Efektywnie wygodne jest powiązanie wszystkich plików związanych z danym apletem w jeden zbiór (np. eliminacja czasu nawiązania połączeń osobno dla każdego pliku HTTP 1.0). Java daje taką możliwość poprzez stworzenie archiwum narzędziem dostarczanym w dystrybucji JDK a mianowicie: JAR. Wywołanie polecenia jar z odpowiednimi przełącznikami i atrybutami umożliwia liczne operacje na archiwach. Przykładowe opcje:

-c stwórz nowe archiwum
-t pokaż zawartość archiwum
-x pobierz podane pliki z archiwum
-u aktualizuj archiwum
-f określenie nazwy archiwum
-O stwórz archiwum bez domyślnej kompresji plików metodą ZIP

W celu stworzenia archiwum z wszystkich klas zawartych w bieżącym katalogu i nadać mu nazwę JediArchiwum można wywołać polecenie:

```
jar  -cf JediArchiwum *.class
```

Tak stworzone archiwum można przejrzeć: jar –tf JediArchiwum, lub odtworzyć: jar –xf JediArchiwum.

W celu wykorzystania apletu, którego kod jest zawarty w pliku archiwum trzeba wykorzystać pole ARCHIVES i nadać mu wartość nazwy pliku archiwum np.: code=Jedi2 archives=JediArchiwum.jar. Należy pamiętać o tym, że w polu code podaje się tylko nazwę klasy apletu bez rozszerzenia *.class.

W celu demonstracji omówionych zagadnień rozbudujmy aplet, a właściwie wywołujący go kod HTML z przykładu 1.4:

Przykład 5.1:

```
// Jedi2.java :

import java.applet.Applet;
import java.awt.*;

public class Jedi2 extends Applet{

        public void paint(Graphics g){
                g.drawString(”Rycerz Luke ma niebieski miecz.„, 15,15);
        }

} // koniec public class Jedi2.class extends Applet

-----------------------------------------------------------------
Jedi2n.html :

<html>
<head>
<title> Przykładowy aplet</title>
</head>

<body>
<applet code=Jedi2.class name= aplecik
width=200 height=100
alt=”Aplet drukujący tekst”
align=middle
hspace=5 vspace=5>
Tu pojawia się aplet
</applet>
</body>
</html>
```

Łatwo zauważyć, że opis i znaczenie pól dla elementu APPLET w HTML jest podobne jak dla elementu IMG. Spośród wymienionych pól konieczne do zastosowania w celu opisu apletu są pola CODE oraz WIDTH i HEIGHT. Te ostatnie dwa są wymagane przez program appletviewer, niemniej w niektórych przeglądarkach istnieje możliwość wprowadzenia opisu apletu bez podania rozmiaru okna apletu. Czy będzie wówczas wygenerowane domyślne okno? Tak, obszarem okna będzie wolne pole edycyjne dostępne przeglądarce (zależne od rozdzielczości ekranu). Oczywiście nie jest to właściwe działanie, i ważne jest ustawianie pól WIDTH oraz HEIGHT.

Poniższy program ukazuje możliwość wykorzystania nazwy apletu oraz pobrania rozmiaru apletu:

Przykład 5.2:

```
//Jedi2nn.java

import java.applet.Applet;
import java.awt.*;

class Sith2n{
	Applet app;
	Sith2n(Applet a){
		this.app=a;
	}
	public Dimension rozmiar(){
		Dimension d = app.getSize();
		return d;
	}

}// koniec class Sith

public class Jedi2n extends Applet{

	public void paint(Graphics g){
		g.drawString("Rycerz Luke ma niebieski miecz.", 15,15);
		Sith2n s = new Sith2n(this);
		Dimension d = s.rozmiar();
		String roz = new String("Szerokość="+d.width+
		"; wysokość="+d.height);
		g.drawString(roz,15,30);
		g.drawString(info(),15,45);
	}
	public String info(){
		Applet aplet=getAppletContext().getApplet("aplecik");
		return (aplet.getCodeBase()).toString();
	}

} // koniec public class Jedi2n extends Applet

//-----------------------------------------------------------
//Jedi2nn.html:

<html>
<head>
<title> Przykładowy aplet</title>
</head>

<p> Oto aplet: </p>
<body>
<applet code=Jedi2n.class name= aplecik
width=200 height=100
alt="Aplet drukujący tekst"
align=middle
hspace=5 vspace=5 >
```

```
Tu pojawia się aplet
</applet>
</body>
</html>
```

W przykładzie tym zastosowano metodę *getSize*() zwracającą obiekt klasy *Dimension* zawierający pola szerokości (width) i wysokość (height) opisujące rozmiar okna apletu. Metoda *info*() przekazuje tekst opisujący adres źródłowy apletu, którego obiekt jest uzyskiwany przez odwołanie się do nazwy apletu. Jest to typowo edukacyjny przykład (metoda *info*() mogła by być` zrealizowana tylko poprzez odwołanie się do *this.getCodeBase()).*

Powyższy kod został rozbity na dwie klasy celem dodatkowego pokazania możliwości pracy z archiwami. W celu stworzenia archiwum składającego się z dwóch klas Jedi2n.class oraz Sith.class należy wywołać program jar:

```
jar -cf JediArchiwum.jar Jedi2n.class Sith2n.class
```

Po stworzeniu archiwum warto sprwdzić jego zawartość poprzez:

```
jar -tf JediArchiwum.jar
```

Tak przygotowane archiwum możemy wykorzystać do konstrukcji kodu HTML wywołującego aplet Jedi2n:

```
//Jedi2nnn.html

<html>
<head>
<title> Przykładowy aplet</title>
</head>

<p> Oto aplet: </p>
<body>
<applet code=Jedi2n name= aplecik archives=JediArchiwum.jar
width=200 height=100
alt="Aplet drukujący tekst"
align=middle
hspace=5 vspace=5 >
Tu pojawia się aplet
</applet>
</body>
</html>
```

Oczywiście efekt działanie jest taki sam jak poprzednio.

Ostatnie pole w opisie apletu, które warto omówić to pole przechowujące parametr apletu. Parametr jest skonstruowany poprzez dwa argumenty: *name* oraz *value*. W polu *name* parametr przechowuje swoją nazwę, natomiast w polu *value* swoją wartość. Konstrukcja parametru jest następująca:
<param name=Nazwa value=Wartość>
np.:

```
<applet code=Jedi2n.class width=300 hight=100>
<param name= Opis value="To jest aplet">
</applet>
```

Dla obu pól parametru istotna jest wielkość znaków. Jeżeli pole wartość przechowuje spację lub znaki specjalne musi być ujęte w cudzysłowie. Ilość parametrów jest dowolna. W kodzie apletu można pobrać wartość parametru o danej nazwie wykorzystując metodę *getParameter*() z argumentem stanowiącym nazwę parametru, np. *getParameter(„Opis").*

Przydatną metodą klasy *Applet* jest metoda *showStatus().* Podawany jako argument tej metody tekst jest wyświetlany w oknie statusu przeglądarki. Jest to często bardzo wygodna metoda powiadamiania użytkownika o działaniu apletu. Omawiane wcześniej (rozdział 1) metody obsługi apletu (*init(), start(), stop(), paint(), destroy())* pozwalają się domyśleć jaką formę przybiera każdy aplet. Otóż każdy aplet jest właściwie wątkiem (podobieństwo start(), stop()) należącym do domyślnej grupy o nazwie applet-tu_nazwa_klasy_z_rozszerzeniem, z domyślnym priorytetem 5 o domyślnej nazwie Thread-tu_kolejny_numer. Metody start() i stop() apletu nie odpowiadają oczywiście metodą o tych samych nazwach zdefiniowanych w klasie Thread. Należy więc pamiętać o tym, że zmiana strony WWW w przeglądarce wcale nie oznacza zakończenia pracy apletu. Zmiana strony oznacza jedynie wywołanie metody stop() apletu czyli żądanie zawieszenia aktywności apletu (np. zatrzymanie animacji) - przejście od statusu *start* do *init*. Standardowo metoda stop() nic nie robi. Jeżeli jest potrzebna jakaś akcja przy zmianie stron wówczas musi być ona zdefiniowana w ciele metody stop(). Podobnie jest z metodą *destroy*(), z tym, że wołana jest ona bezpośrednio przez zakończeniem pracy apletu (np. koniec pracy przeglądarki). Omawiane metody obsługi apletu są więc wywoływane przez aplikację kontrolującą aplet w wyniku wystąpienia określonych zdarzeń, i mogą być przedefiniowane przez programistę celem wprowadzenia zamierzonego działania.

Należy tu podkreślić bardzo ważny element w korzystaniu z przeglądarek w pracy z apletami. Aby uruchomić aplet dana przeglądarka musi być zgodna, tzn., dostarczać Maszynę Wirtualną do obsługi kodu bajtów. Obsługiwana wersja Javy może być łatwo uzyskana poprzez uruchomienie konsoli Javy, którą stanowi opcjonalne okno przeglądarki. W celu uniezależnienia się od zaimplementowanej wersji Javy w danej przeglądarce Sun opracował i udostępnia plug-in Javy. Po zainstalowaniu wtyczki i odpowiednim przygotowaniu kodu HTML (*HTMLconverter*) można uruchomić praktycznie każdy dobrze napisany aplet, nawet dla najnowszej wersji JDK obsługiwanej zazwyczaj przez plug-in. Przygotowanie kodu HTML zawierającego aplet do pracy z wtyczką wymaga znacznych zmian w opisie apletu. Jest to spowodowane tym, że w części kodu HTML gdzie był uruchamiany aplet musi być uruchamiana wtyczka, której parametrami wywołania są kod klasy apletu, nazwa apletu oraz typ elementu. Dotychczasowe wywołanie apletu przez tag <APPLET> musi być usunięte lub wzięte w komentarz. Pomocnym narzędziem wykonującym automatyczną konwersję kodu HTML do wersji zgodnej z wtyczką jest aplikacja Javy: *HTMLConverter*. Sun dostarcza tę aplikację w celu szybkiego dostosowania kodu HTML, bez konieczności ręcznej modyfikacji źródła HTML. *HTMLConverter* może dokonać konwersji pojedynczego pliku HTML lub całej serii plików wykonując przy tym kopie zapasowe wersji oryginalnych w jednym z katalogów. Przykładowa treść

pliku HTML prezentowanego wcześniej (Jedi2n.html) po konwersji będzie wyglądała następująco:

```
//Jedi2n.html po konwersji

<html>
<head>
<title> Przykładowy aplet</title>
</head>

<body>
<p> To jest aplet </p>
<!--"CONVERTED_APPLET"-->
<!-- CONVERTER VERSION 1.0 -->
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 200 HEIGHT = 100 NAME = aplecik ALIGN = middle VSPACE = 5 HSPACE = 5 ALT = "Aplet drukujący tekst"  codebase="http://java.sun.com/products/plugin/1.2/jinstall-12-win32.cab#Version=1,2,0,0">
<PARAM NAME = CODE VALUE = Jedi2.class >
<PARAM NAME = NAME VALUE = aplecik >

<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.2" java_CODE = Jedi2.class ALT = "Aplet drukujący tekst" NAME = aplecik WIDTH = 200 HEIGHT = 100 ALIGN = middle VSPACE = 5 HSPACE = 5 pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html"><NOEMBED></COMMENT>

</NOEMBED></EMBED>
</OBJECT>

<!--
<APPLET  CODE = Jedi2.class WIDTH = 200 HEIGHT = 100 NAME = aplecik ALIGN = middle VSPACE = 5 HSPACE = 5 ALT = "Aplet drukujący tekst" >

</APPLET>
-->
<!--"END_CONVERTED_APPLET"-->

<p> Fajny no nie </p>
</body>
</html>
```

Wyraźnie widać podział pliku na część związaną z tagiem OBJCET oraz na część wziętą w komentarz związaną z tagiem APPLET. Oczywiście aby można była skorzystać z tego kodu należy zainstalować wtyczkę do przeglądarki. Kod instalacyjny można pobrać z serwera opisanego w polu pluginspage równym "http://java.sun.com/products/plugin/1.2/plugin-install.html.

Powyżej omówione zostały podstawowe zagadnienia związane tylko z apletami. Bardzo ważne są również inne tematy dotyczące apletów jak np. wykorzystanie grafiki, bezpieczeństwo, itp., które będą systematycznie w tych materiałach omawiane.

5.2 Grafika w Javie

Biblioteka Abstract Window Toolkit - AWT jako historycznie pierwsza w JDK dostarcza szereg elementów wykorzystywanych do tworzenia interfejsów graficznych dla potrzeb komunikacji z
użytkownikiem i obsługi jego danych. Do najbardziej istotnych elementów tego pakietu należy zaliczyć:

komponenty (Components {widgets}),
rozkłady (Layout managers),
zdarzenia (Events),
rysunki i obrazy (Drawings and images).

Od wersji JAVA JDK 2 wprowadzono istotne rozszerzenia AWT związane z przygotowaniem GUI. Podstawowe elementy tej rozbudowy to pakiety *javax.swing* i tak zwany zestaw Java2D rozszerzający klasy biblioteki *java.awt*. Do najbardziej istotnych elementów *javax.swing* i Java2D należy zaliczyć:
- nowe, rozbudowane graficznie komponenty w SWING,
- nowe rozkłady i metody pracy z kontenerami,
- rozbudowa biblioteki *java.awt.image*,
- dodanie nowych klas do biblioteki *java.awt* (np. Graphics2D).

Platforma 2 Javy w sposób znaczny zmienia możliwości tworzenia aplikacji stąd krótko zostaną omówione podstawowe elementy pakietu AWT i SWING wraz z
ich porównaniem.

5.2.1 Komponenty

Komponenty to podstawowe elementy graficzne aktywne lub bierne służące do tworzenia interfejsu graficznego. Komponenty są reprezentowane przez klasy głównie dziedziczące z klasy *Component:*
Box.Filler, Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent

Każdy komponent posiada odpowiednio skonstruowaną klasę i metody, które opisane są w dokumentacji API.
Klasa *Component* stanowi klasę abstrakcyjną tak więc nie odwołuje się do niej bezpośrednio (przez stworzenie obiektu za pomocą *new*) lecz przez klasy potomne. Klasa *Component* dostarcza szereg ogólnie wykorzystywanych metod np.:

public Font getFont() - zwraca informacje o czcionce wybranej dla komponentu
public Graphics getGraphics() - zwraca kontekst graficzny komponentu potrzebny przy wywoływaniu metod graficznych
public void paint(Graphics g) - rysuje komponent,
itp.

Jednym z komponentów jest kontener reprezentowany przez klasę Container. Kontener jest komponentem, w którym można umieszczać inne komponenty.

Przykładowym kontenerem wykorzystywanym dotąd w tym materiale w przykładach jest *Applet* (dziedziczy z klasy Panel, a ta z klasy *Container*). Podstawowe klasy (kontenery) dziedziczące z klasy *Container* to:
BasicSplitPaneDivider, Box, CellRendererPane, DefaultTreeCellEditor.EditorContainer, JComponent, Panel, ScrollPane, Window

Najczęściej wykorzystywane kontenery w AWT to *Panel* oraz *Window*. Ten ostatni nie jest wprost wykorzystywany lecz poprzez swoje podklasy:
BasicToolBarUI.DragWindow, Dialog, Frame, JWindow

Klasa *Frame* jest kontenerem bezpośrednio wykorzystywanym do tworzenia okien graficznych popularnych w GUI. *Dialog* jest kontenerem dedykowanym komunikacji z użytkownikiem i stanowi okno o zmienionej funkcjonalności względem *Frame* (brak możliwości dodania paska Menu).

Podstawowa praca w konstrukcji interfejsu graficznego w Javie polega na:
- zadeklarowaniu i zdefiniowaniu kontenera,
- zadeklarowaniu komponentu,
- zdefiniowaniu (zainicjowanie) komponentu
- dodanie komponentu do kontenera.

Ostatni krok w konstrukcji interfejsu związany jest ze sposobem umieszczania komponentów w kontenerze. Sposób te określany jest w Javie rozkładem (Layout). Z każdym kontenerem jest więc związany rozkład, w ramach którego umieszczane są komponenty.

Zanim omówione zostaną kontenery i rozkłady warto zapoznać się z typami komponentów, które w tych kontenerach mogą być umieszczane.
Do podstawowych komponentów należy zaliczyć:
a.) dziedziczące pośrednio i bezpośrednio z klasy *Component*:
- *Label* – pole etykiety
- *Button* – przycisk
  Canvas – pole graficzne
  Checkbox – element wyboru (logiczny)
  Choice -element wyboru (z listy)
  List – lista elementów
  Scrollbar – suwak
  TextComponent:
    TestField: pole tekstowe
    TextArea: obszar tekstowy

b) nie dziedziczące z klasy *Component*:
    MenuBar – pasek menu,
    MenuItem - element menu:
        Menu – menu (zbiór elementów)
        PopupMenu – menu podręczne.

Komponent, którego klasa nie dziedziczy z klasy *Component* to *MenuComponent*. Klasa ta jest nadklasą komponentów: *MenuBar*, *MenuItem*. Ta ostatnia dostarcza również poprzez klasy dziedziczące z niej następujące komponenty: Menu oraz *PopupMenu*. Łącznie cztery klasy tj. *MenuBar*, *MenuItem*, Menu oraz *PopupMenu* są

wykorzystywane do tworzenia menu programu graficznego.

Wszystkie komponenty biblioteki AWT są dobrze opisane w dokumentacji (Java API) Javy, i dlatego nie będą szczegółowo omawiane w tym materiale. Rozpatrzmy jedynie metodę tworzenia komponentu i dodawania go do kontenera. Jedynym kontenerem poznanym do tej pory był *Applet* i dlatego będzie on wykorzystany w poniższych przykładach.

Przykładowy komponent - *Button* - umożliwia stworzenie przycisku graficznego wraz z odpowiednią etykietą tekstową. Stworzenie obiektu typu *Button* polega na implementacji prostej instrukcji:

```
Button przyciskKoniec = new Button("Koniec");
```

Dodanie tak stworzonego elementu do tworzonego interfejsu wymaga jeszcze wywołania metody:

```
add(przyciskKoniec);
```

Po uaktywnieniu interfejsu przycisk będzie widoczny dla użytkownika. W podobny sposób tworzy się obiekty pozostałych klas komponentów, manipuluje się ich metodami dla uzyskania odpowiedniego ich wyglądu, oraz dodaje się je do tworzonego środowiska graficznego.

Przykładowy kod źródłowy dla komponentu *Button* może wyglądać w następujący sposób:

Przykład 5.3:

```
//Ognia.java:

import java.awt.*;
import java.applet.*;

public class Ognia extends Applet {

            public void init() {
               Button przyciskTest = new Button("przycisnij mnie...");
               add(przyciskTest);
            }
}// koniec public class Ognia

//************************************

Ognia.html :
...
<APPLET  CODE="Ognia.class" WIDTH=70 HEIGHT=40 ALIGN=CENTER></APPLET>
```

Praca z pozostałymi komponentami wygląda podobnie. Poniższy przykład demonstruje różne typy komponentów.

Przykład 5.4:

```
//Pulpit.java:

public class Pulpit extends Applet {

        public void init() {
                Button przyciskTest = new Button("ognia...");
                 add(przyciskTest);

                 Label opis = new Label("Strzelac");
                add(opis);

                Checkbox rak = new Checkbox("Rakieta");
                Checkbox bomb = new Checkbox("Bomba");
                add(rak);
                add(bomb);

                Choice kolor = new Choice();
                kolor.add("zielona");
              kolor.add("czerwona");
              kolor.add("niebieska");
                add(kolor);

                List lista = new List(2, false);
                  lista.add("mała");
                  lista.add("malutka");
                  lista.add("wielka");
                  lista.add("duża");
                  lista.add("ogromna");
                  lista.add("gigant");
                add(lista);

                TextField param = new TextField("Podaj parametry");
                add(param);

        }
}// koniec public class Pulpit

//*******************************************************

<html>
<APPLET  CODE="Pulpit.class" WIDTH=800 HEIGHT=80 ALIGN=CENTER></APPLET>
</html>
```

Komponenty związane z menu są często wykorzystywane przy konstrukcji GUI, niemniej wymagają kontenera typu *Frame*, dlatego zostaną omówione później (wraz z tym kontenerem).

Pakiet *javax.swing* dostarcza szeregu zmodyfikowanych i nowych kontenerów i komponentów. Między innymi zmodyfikowano system umieszczania komponentów w kontenerach (co jest omówione dalej) oraz dodano możliwość wyświetlania ikon na komponentach. Przykładowe komponenty z pakietu *javax.swing* można znaleźć w dokumentacji SWING. Wszystkie komponenty tej biblioteki są reprezentowane przez

odpowiednie klasy, których nazwa zaczyna się od J, np. *JButton*. Komponenty biblioteki SWING dziedziczą z AWT (np. z klasy *Component*). Podstawowe komponenty tej biblioteki są podobne jak dla AWT z tym, że oznaczane z dodatkową literą J. Wprowadzono również nowe i przeredagowane elementy jak np.:

JColorChooser – pole wyboru koloru
JFileChooser – pole wyboru pliku
JPasswordField – pole hasła,
JProgressBar – pasek stanu
JRadioButton – element wyboru
JScrollBar - suwak
JTable - tabela
JTabbedPane - pole zakładek
JToggleButton –przycisk dwu stanowy
JToolBar – pasek narzędzi
JTree – lista w postaci drzewa

Implementacja komponentów SWING jest podobna do implementacji komponentów zdefiniowanych w AWT.

Przykład 5.5:

```
//Przycisk.java:

import java.awt.*;
import java.applet.*;
import javax.swing.*;

public class Przycisk extends JApplet {

        public void init() {
                String s. = getCodeBase().toString()+”/”+”jwr2.gif”;
                /* dla apletu uruchamianego z serwera HTTP
                w przeciwnym wypadku może nastąpić błąd bezpieczeństwa
                ➔ próba odczytu z dysku */
                JButton przyciskTest = new JButton(new ImageIcon(s));
                 getContentPane().add(przyciskTest);


        }
}// koniec public class Przycisk
```

W powyższym przykładzie zastosowano kontener z biblioteki SWING - *JApplet*, i dlatego inna jest forma dodania tworzonego komponentu (*JButton*) do tego kontenera (*getContentPane().add*()). Różnica ta zostanie opisana w podrozdziale dotyczącym rozkładów.

5.2.2 Kontenery

Do tej pory przedstawione zostały dwa kontenery (komponenty zawierające inne komponenty): *Applet* oraz *JApplet*. Warto zapoznać się z pozostałymi.

Zasadniczym kontenerem jest w Javie jest okno reprezentowane przez klasę *Window*. Kontener taki nie posiada ani granic okna (ramy) ani możliwości dodania menu. Stanowi więc jakby pole robocze. Dlatego nie korzysta się z tego kontenera wprost, lecz poprzez klasy potomne: *Frame* oraz Dialog. Obiekt klasy *Frame* jest związany z kontenerem okna graficznego o sprecyzowanych cechach (ramie - ang. *Frame*). Kontener taki stanowi więc to, co jest odbierane w środowisku interfejsu graficznego przez okno graficzne. Stąd w Javie przyjęło się określać kontener klasy *Frame* mianem okna. Okno może posiadać pasek menu. Drugim kontenerem reprezentowanym przez klasę dziedziczącą z klasy *Window* - jest kontener typu *Dialog*. Kontener ten podobnie jak okno graficzne (*Frame*) posiada określone ramy, lecz różni się od tego okna tym, że nie można mu przypisać paska menu. Okno dialogowe (kontener reprezentowany przez klasę *Dialog*) posiada zawsze właściciela, jest zatem zależne. Inne, często wykorzystywane kontenery to Panel oraz *ScrollPane*. Panel służy do wyodrębnienia w kontenerze głównym kilku obszarów roboczych. Klasa Panel nie posiada więc prawie w ogóle żadnych metod (1) własnych. *ScrollPane* jest kontenerem umożliwiającym wykorzystanie pasków przesuwu poziomego i pionowego dla kontenera głównego. Oczywiście biblioteka SWING definiuje własne kontenery, których nazwa zaczyna się od J, i tak np. *JFrame, JWindow, JDialog, JPanel, JApplet, JFileDialog*, itp.
Znając charakterystykę kontenerów warto zapoznać się z możliwością ich wykorzystania. Zasada jest bardzo prosta - kontenery tworzy się dla:

1. aplikacji - zasadniczy kontener to *Frame* lub *JFrame*,
2. apletu - zasadniczy kontener to *Applet* lub *JApplet*.

Nie oznacza to jednak, że nie można używać okien w apletach i apletów w oknach!
Tworząc okno graficzne należy kierować się następującą procedurą:

1. deklaracja i zainicjowanie okna, np.: Frame okno = new Frame(„Program”);
2. ustawienie parametrów okna, np. okno.setSize(300,300);
3. dodanie komponentów do okna, np.: okno.add(new Button(„Ognia”));
4. ustawienie okna jako aktywnego, np.: okno.setVisible(true);

Brak ostatniego kroku spowoduje, że zdefiniowane okno będzie niewidoczne dla użytkownika. Rozpatrzmy następujący przykład:

Przykład 5.6:

```
//Dzialo.java:

import java.awt.*;

class Okno extends Frame{
	Okno(String nazwa){
		super(nazwa); //metoda super wywołuje konstruktor nadklasy
		setResizable(false);
		setSize(400,400);
	}
}// koniec class Okno

public class Dzialo{
```

```
	public static void main(String args[]){
		Okno o = new Okno("Panel sterujący działem");
		o.add(new Button("Ognia"));
		o.setVisible(true);
	}
}// koniec class Dzialo
```

Efektem działania kodu będzie stworzenie okna graficznego o niezmiennych rozmiarach 400/400 pikseli. Okno zawiera jeden przycisk pokrywający cały obszar roboczy. Ponieważ nie omówiono jeszcze metod obsługi zdarzeń nie możliwe jest zakończenie pracy okna przez wykorzystanie kontrolek okna. Dlatego trzeba przerwać pracę aplikacji w konsoli (CTRL-C). Okno (podobnie jak Aplet) może być podzielone na obszary poprzez wykorzystanie paneli. Co więcej panele mogą również być podzielone na inne panele. Powyższy kod można rozwinąć do postaci:

Przykład 5. 7:

```
//DzialoS.java:

import java.awt.*;

class Okno extends Frame{
	Okno(String nazwa){
		super(nazwa);
		setResizable(false);
		setSize(400,400);
	}
}// koniec class Okno

public class DzialoS{

	public static void main(String args[]){
		Okno o = new Okno("Panel sterujący działem");
		o.setLayout(new FlowLayout()); //zmiana rozkładu
		Panel p = new Panel();
		p.add(new Button("Ognia"));
		p.add(new Button("Stop"));
		o.add(p);
		Panel p1 = new Panel();
		p.add(new Label("Kontrola działa"));
		o.add(p1);
		o.setVisible(true);
	}
}// koniec class DzialoS
```

W kodzie powyższym wprowadzono dwa panele dzielące okno na dwie części. W panelu pierwszy umieszczono dwa przyciski, natomiast w panelu drugim jedną etykietę. Bezpośrednio po zainicjowaniu obiektu okna dokonano zmiany metody rozkładu elementów aby umożliwić wyświetlenie wszystkich tworzonych komponentów. Rozkłady są omówione dalej. Omawiany przykład wskazuje na ciekawy sposób konstrukcji interfejsu graficznego w Javie. Praktycznie interfejs składany jest z klocków (kontenery i komponenty) co znacznie upraszcza procedurę

tworzenia programów graficznych.
Z kontenerem typu *Frame* związany jest zbiór komponentów menu. W celu stworzenia menu okna graficznego należy wykonać następujące kroki:

1. zainicjować obiekt klasy *MenuBar* reprezentujący pasek menu;
2. zainicjować obiekt klasy Menu reprezentujący jedną kolumnę wyborów,
3. zainicjować obiekt klasy *MenuItem* reprezentujący element menu w danej kolumnie
4. dodać element menu do Menu;
5. powtórzyć kroki 2,3,4 tyle razy ile ma być pozycji w kolumnie i kolumn
6. dodać pasek menu do okna graficznego.

Realizację menu przedstawia następujący program:

Przykład 5. 8:

```
//DzialoM.java:

import java.awt.*;

class Okno extends Frame{
	Okno(String nazwa){
		super(nazwa);
		setResizable(false);
		setSize(400,400);
	}
}// koniec class Okno

public class DzialoM{

	public static void main(String args[]){
		Okno o = new Okno("Panel sterujący działem");
		MenuBar pasek = new MenuBar();

		Menu plik = new Menu("Plik");
		plik.add(new MenuItem("-")); //separator
		plik.add(new MenuItem("Ognia"));
		plik.add(new MenuItem("Stop"));
		pasek.add(plik);

		Menu edycja = new Menu("Edycja");
		edycja.add(new MenuItem("-"));
		edycja.add(new MenuItem("Pokaż dane działa"));
		pasek.add(edycja);

		o.setMenuBar(pasek);

		o.add(new Button());
		o.setVisible(true);

		Dialog d = new Dialog(o,"Dane działa", false);
		d.setSize(200,200);
		d.setVisible(true);
```

```
            d.show();
      }
}// koniec class DzialoM
```

Przykład ten demonstruje mechanizm tworzenia menu w oknie graficznym. Warto zauważyć, że dodanie paska menu do okna odbywa się nie poprzez metodę *add*() lecz przez *setMenuBar*(). Jest to istotne ze względu na ukazanie różnicy pomiędzy komponentem typu menu a innymi komponentami, które dziedziczą z klasy *Component* jak *Button, Label, Panel*. W omawianym przykładzie ukazano również metodę tworzenia okien dialogowych. Wywołany konstruktor powoduje utworzenie okna dialogowego właścicielem którego będzie okno graficzne „o”; nazwą okna dialogowego będzie „Dane działa” oraz okno to nie będzie blokować operacji wejścia.

W pakiecie SWING również zdefiniowano klasy komponentów związane z menu. Jak w większości przypadków, główna różnica pomiędzy analogicznymi komponentami AWT i SWING polega na tym, że te ostatnie umożliwiają implementację elementów graficznych (ikon). Możliwe jest więc na przykład skonstruowanie menu, w którym każdy element jest reprezentowany tylko przez ikonę graficzną.

5.2.3 Rozkłady

Bardzo częstym dylematem programisty jest problem rozkładu komponentów w ramach tworzonego interfejsu graficznego. Problem rozdzielczości ekranu, liczenia pikseli to typowe zadania do rozwiązania przed przystąpieniem do projektu interfejsu. W języku JAVA problemy te w znaczny sposób zredukowano wprowadzając tzw. rozkłady. Rozkład oznacza nic innego jak sposób układania komponentów na danej formie (np. aplet , panel). System zarządzający rozkładami umieszcza dany komponent (jako rezultat działania metody *add*()) zgodnie z przyjętym rozkładem. Definiowane w AWT rozkłady to:

BORDER LAYOUT - (domyślny dla kontenerów*: Window, Frame, Dialog, JWindow, JFrame, JDialog*) komponenty są umieszczane i dopasowywane do pięciu regionów: północ, południe, wschód, zachód oraz centrum. Każdy z pięciu regionów jest identyfikowany przez stałą z zakresu: NORTH, SOUTH, EAST, WEST, oraz CENTER.

CARD LAYOUT - każdy komponent w danej formie (kontenerze) jest rozumiany jako karta. W danym czasie widoczna jest tylko jedna karta, a forma jest rozumiana jako stos kart. Metody omawianej klasy umożliwiają zarządzanie przekładaniem tych kart.

FLOW LAYOUT - (domyślny dla kontenerów*: Panel, Applet, JPanel, JApplet*) komponenty są umieszczane w ciągu "przepływu" od lewej do prawej (podobnie do kolejności pojawiania się liter przy pisaniu na klawiaturze).

GRID LAYOUT - komponenty są umieszczane w elementach regularnej siatki (gridu). Forma (kontener) jest dzielona na równe pola, w których kolejno umieszczane są komponenty.

GRIDBAG LAYOUT - komponenty umieszczane są w dynamicznie tworzonej siatce regularnych pól, przy czym komponent może zajmować więcej niż jedno pole.

Przykład zastosowania rozkładu BORDER LAYOUT ukazano poniżej:

Przykład 5.9 :

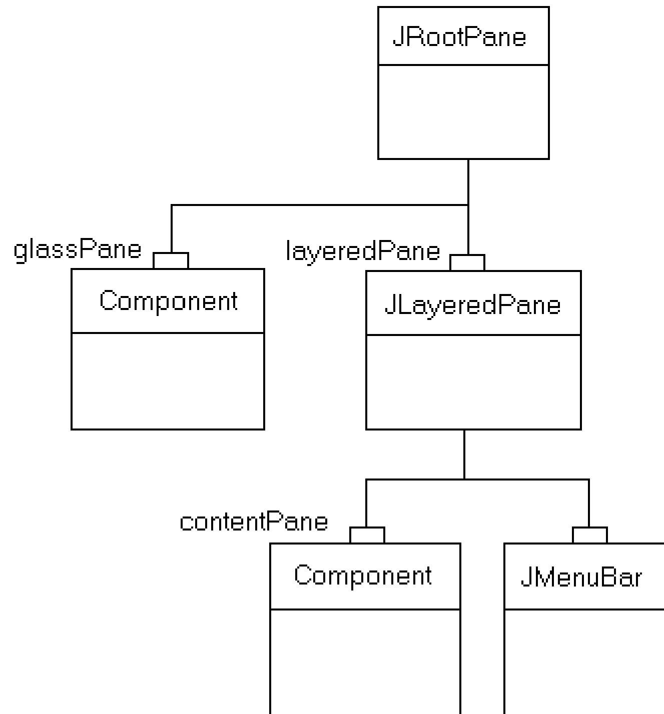
```
//ButtonTest4.java

import java.awt.*;
import java.applet.Applet;

public class ButtonTest4 extends Applet {
	public void init() {
	  setLayout(new BorderLayout());
	  add(new Button("North"), BorderLayout.NORTH);
	  add(new Button("South"), BorderLayout.SOUTH);
	  add(new Button("East"), BorderLayout.EAST);
	  add(new Button("West"), BorderLayout.WEST);
	  add(new Button("Center"), BorderLayout.CENTER);
	}
} // koniec public class ButtonTest4
```

Pakiet Swing wprowadzający nowe kontenery i komponenty ustala również nowe metody używania rozkładów oraz wprowadza nowe rozkłady. Kontener (forma) zawiera podstawowe komponenty zwane PANE (płyty).

Podstawowym komponentem jest tu płyta główna (korzeń) - *JRootPane*. Płyta ta jest fundamentalną częścią kontenerów w Swing takich jak *JFrame*, *JDialog*, *JWindow*, *JApplet*. Oznacza to, że umieszczenie komponentów nie odbywa się bezpośrednio przez odwołanie do tych kontenerów lecz poprzez *JRootPane*. Dla potrzeb konstrukcji GUI z wykorzystaniem elementów pakietu Swing istotna jest znajomość struktury *JRootPane*, pokazanej poniżej.



Rysunek 5.2 Konstrukcja płyty rootPane

JRootPane jest tworzona przez *glassPane* (szyba, płyta szklana) i *layeredPane* (warstwę płyt), na którą składają się: opcjonalna *menuBar* (pasek menu) oraz *contentPane* (płyta robocza, zawartość). Płyta *glassPane* jest umieszczona zawsze na wierzchu wszystkich elementów (stąd szyba) i jest związana z poruszaniem myszy. Ponieważ *glassPane* może stanowić dowolny komponent możliwe jest więc rysowanie w tym komponencie. Domyślnie *glassPane* jest niewidoczna. Płyta *contentPane* stanowi główną roboczą część kontenera gdzie rozmieszczamy komponenty i musi być zawsze nadrzędna względem każdego "dziecka" płyty *JRootPane*. Oznacza to, że zamiast bezpośredniego dodania komponentu do kontenera musimy dodać element do płyty *contentPane*:

```
- zamiast dodania:
        rootPane.add(child);
     czy jak to było w AWT np.:
         Frame frame;
         frame.add(child);

- wykonujemy:
        rootPane.getContentPane().add(child);
      np.:
         JFrame frame;
         frame.getContentPane().add(child);
```

Jest to niezwykle ważna różnica pomiędzy klasycznym dodawaniem elementów do kontenerów w AWT a w SWING. Oznacza to, że również ustawienia

rozkładów muszą odwoływać się do płyt zamiast bezpośrednio do kontenerów np. frame.getContentPane().setLayout(new GridLayout(3,3)).

Dodatkowo w pakiecie Swing wprowadzono inne rozkłady takie jak:

- BOX LAYOUT,
- OVERLAY LAYOUT,
- SCROLLPANE LAYOUT,
- VIEWPORT LAYOUT.

Przykładowo można porównać działanie rozkładu *BorderLayout* dla implementacji appletu w AWT i SWING:

Przykład 5.10 :

```
// AWT:
// ButtonTest4.java

import java.awt.*;
import java.applet.Applet;

public class ButtonTest4 extends Applet {
        public void init() {
          setLayout(new BorderLayout());
          add(new Button("North"), BorderLayout.NORTH);
          add(new Button("South"), BorderLayout.SOUTH);
          add(new Button("East"), BorderLayout.EAST);
          add(new Button("West"), BorderLayout.WEST);
          add(new Button("Center"), BorderLayout.CENTER);
        }
} // koniec public class ButtonTest4
```

Przykład 5.11 :

```
// SWING:
// ButtonTest5.java:

import java.awt.*;
import java.applet.Applet;
import javax.swing.*;

public class ButtonTest5 extends JApplet {
        public void init() {
          getContentPane().setLayout(new BorderLayout());
          getContentPane().add(new Button("North"), BorderLayout.NORTH);
          getContentPane().add(new Button("South"), BorderLayout.SOUTH);
          getContentPane().add(new Button("East"), BorderLayout.EAST);
          getContentPane().add(new Button("West"), BorderLayout.WEST);
          getContentPane().add(new Button("Center"), BorderLayout.CENTER);
        }
} // koniec public class ButtonTest5
```

W celu zrozumienia różnicy warto skompilować i uruchomić aplet buttonTest5 z usunięciem części " *getContentPane*()" z *linii getContentPane().setLayout(new BorderLayout())*; w kodzie programu.

5.2.4 Zdarzenia

Oprócz wyświetlenia komponentu konieczne jest stworzenie odpowiedniego sterowania związanego z danym komponentem. W przykładowym kodzie apletu ButtonTest.java ukazano implementację metody action():

Przykład 5.12:

```
//ButtonTest.java:

import java.awt.*;
import java.applet.*;

public class ButtonTest extends Applet {

                public void init() {
                   Button przyciskTest = new Button("przycisnij mnie...");
                   add(przyciskTest);
                }

                public boolean action(Event e, Object test) {
                   if(test.equals("przycisnij mnie..."))
                     System.out.println("Przycisnieto " + test);
                  else
                    return false;
                  return true;
                }
}// koniec public class ButtonTest

//-------------------------------------------------

ButtonTest.html :
...
<APPLET  CODE="ButtonTest.class" WIDTH=70 HEIGHT=40 ALIGN=CENTER></APPLET>
```

Uwaga! Kompilacja tego kodu w środowisku JDK późniejszym niż 1.0 zgłosi ostrzeżenie o użyciu metody (*action()*) o obniżonej wartości (*deprecated*). Nie powoduje to jednak w tym przypadku przerwania procesu kompilacji.

Metoda ta obsługuje zdarzenia związane z obiektem typu Button, i w rozpatrywanym przykładzie wyświetla tekst na standardowym urządzeniu wyjścia. Warto zwrócić uwagę na instrukcję warunkową if(test.equals()). Otóż metoda action może obsługiwać kilka komponentów i w zależności od obiektu test wykonywać to działanie, które programista umieścił w odpowiedniej części warunkowej. W przypadku obsługi wielu różnych komponentów bardzo często pojawiają się błędy wynikające z nazewnictwa etykiety komponentu (przestawione litery, wielkie litery, itp.). Ogólnie biorąc każdy komponent może wywołać określone zdarzenie (Event),

którego obsługa umożliwia interaktywną pracę z interfejsem programu. W tradycyjnym modelu GUI program generuje interfejs graficzny, a następnie w nieskończonej pętli czeka na pojawienie się zdarzeń, które należy w celowy sposób obsłużyć. W Javie 1.0 obsługę zdarzeń wykonuje się poprzez odpowiednie instrukcje warunkowe określające typ zdarzenia w ciele metod: action() oraz handleEvent() (od JDK 1.1 metody te są wycofywane – „deprecated"). Takie rozwiązanie jest mało efektywne, a na pewno nie jest obiektowe. Od wersji Javy 1.1 wprowadza się nowy system przekazywania i obsługi zdarzeń określający obiekty jako nasłuchujące zdarzeń (listeners) i jako generujące zdarzenia (sources). W tym nowym modelu obsługi zdarzeń komponent "odpala" ("fire") zdarzenie. Każdy typ zdarzenia jest reprezentowany przez określoną klasę. W nowym systemie można nawet stworzyć własne typy zdarzeń. Wygenerowane przez komponent zdarzenie jest odbierane przez właściwy element nasłuchu związany z komponentem generującym zdarzenie. Jeżeli w ciele klasy nasłuchu danych zdarzeń istnieje metoda obsługi tego zdarzenia, to zostanie ona wykonana. Generalnie więc można rozdzielić źródło zdarzeń i obsługę zdarzeń. Najczęstszym modelem programu wykorzystywanym do tworzenia obsługi zdarzeń jest implementacja obsługi zdarzeń jako klasa wewnętrzna (inner class) klasy komponentu, którego zdarzenia mają
być obsługiwane. Przykładowo:

Przykład 5.13:

```
//ButtonTest2.java:

import java.awt.*;
import java.awt.event.*; // Koniecznie należy pamiętać o importowaniu pakietu event
import java.applet.*;

public class ButtonTest2 extends Applet {
  Button b1, b2;
  public void init() {
    b1 = new Button("Przycisk 1");
    b2 = new Button("Przycisk 2");
    b1.addActionListener(new B1());  //dodajemy obiekt nasłuchujący zdarzenia dla komponentu b1
    b2.addActionListener(new B2());  //dodajemy obiekt nasłuchujący zdarzenia dla komponentu b1
    add(b1);
    add(b2);
  } // koniec public void init()

  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      getAppletContext().showStatus("Przycisk 1");
    }
  } // koniec class B1

  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      getAppletContext().showStatus("Przycisk 2");
    }
  } // koniec class B2
  /* Dla porównania stary model obsługi zdarzeń:
  public boolean action(Event e, Object test) {
    if(e.target.equals(b1))  //lub test.equals("Przycisk 1"); gorsze rozwiazanie
```

```
      getAppletContext().showStatus("Przycisk 1");
   else if(e.target.equals(b2))
           getAppletContext().showStatus("Przycisk 2");
        else
           return super.action(e, test);
   return true;
 }
  */

} // koniec public class ButtonTest2 extends Applet

//-------------------------------------------------

ButtonTest2.html :
...
<APPLET  CODE="ButtonTest2.class" WIDTH=70 HEIGHT=40 ALIGN=CENTER></APPLET>
```

Jak widać na prezentowanym przykładzie element nasłuchu zdarzeń jest obiektem klasy implementującej interfejs nasłuchu. To, co musi zrobić programista to stworzyć odpowiedni obiekt nasłuchu dla komponentu odpalającego zdarzenie. Przesłanie obiektu nasłuchu polega na wykonaniu metody addXXXXXListener() dla danego komponentu, gdzie XXXXX oznacza typ zdarzenia jakie ma być nasłuchiwane (np. addMouseListener, addActionListener, addFocusListener, ...).
Sposób rejestracji i nazwa metody obsługującej tę rejestrację daje programiście informację jaki typ zdarzeń jest obsługiwany w danym fragmencie kodu.

Często rejestracja obiektu obsługi zdarzeń zawiera definicję klasy i metody obsługi zdarzenia. W przeciwieństwie do wyżej omawianej metody definiowane klasy są anonimowe. To alternatywne rozwiązanie obsługi zdarzeń jest często wykorzystywane szczególnie tam, gdzie kod obsługi zdarzenia jest krótki. Najprostszym przykładem może być stworzenie ramki - FRAME (okna), która będzie wyświetlana na ekranie i prostego kodu obsługującego zdarzenie zamknięcia ramki (np. metodą System.exit(0);). Standardowo wykorzystuje się następujący kod:

```
...
addWindowListener(new WindowAdapter(){                    //dodajemy obsługę
zdarzeń okna

   public void windowClosing(WindowEvent e){
   //implementacja metody zamykania okna przy wystąpieniu danego zdarzenia
      System.out.println("Dziekujemy za prace z programem...");
  //akcja podejmowana przy zamykaniu okna
      System.exit(0);                                     //koniec programu
  }

});
...
```

Możemy więc teraz stworzyć przykładowy program (może być wywołany zarówno jako aplet jak i aplikacja) demonstrujący omówione zagadnienia:

Przykład 5.14:

```
//ButtonTest3.java:

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ButtonTest3 extends Applet {
  Button b1, b2;
  TextField t = new TextField(20);

  public void init() {
    b1 = new Button("Przycisk 1");
    b2 = new Button("Przycisk 2");
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    add(b1);
    add(b2);
    add(t);
  }

  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Przycisk 1");
    }
  }// koniec class B1

  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Przycisk 2");
    }
  }// koniec class B2

  // statyczna metoda  main() wymagana dla aplikacji:

  public static void main(String args[]) {
    ButtonTest3 applet = new ButtonTest3();
    Frame okno = new Frame("ButtonTest3");
    okno.addWindowListener(new WindowAdapter(){
          public void windowClosing(WindowEvent e){
                          System.out.println("Dziekujemy za prace z programem...");
                          System.exit(0);
          }
   });
    okno.add(applet, BorderLayout.CENTER);
    okno.setSize(300,200);
    applet.init();
    applet.start();
    okno.setVisible(true);
  }
} // koniec public class ButtonTest3 extends Applet
```

W celu właściwej obsługi zdarzeń konieczna jest wiedza dotycząca typu zdarzeń . Związane jest z tym poznanie podstawowych klas zdarzeń. W wersji JDK 1.0 podstawową klasą przechowującą informacje związaną z danym zdarzeniem była klasa java.awt.Event. W celu kompatybilności z JDK1.0 klasa ta występuje również w późniejszych wersjach JDK. Począwszy od JDK 1.1 podstawową klasą zdarzeń jest klasa java.awt.AWTEvent będąca korzeniem wszystkich klas zdarzeń w AWT. Klasa ta dziedziczy z klasy java.util.EventObject będącą korzeniem wszystkich klas zdarzeń w Javie (nie tylko AWT ale np. dla SWING, itp.). Klasy zdarzeń dla AWT są zdefiniowane w pakiecie java.awt.event.* i dlatego w celu obsługi zdarzeń konieczne jest importowanie tego pakietu:
(java.awt.event.*):

AdjustmentEvent
ContainerEvent,
FocusEvent,
KeyEvent,
MouseEvent,
ComponentEvent
Text Event
WindowEvent
ItemEvent
ActionEvent
InputMethodEvent
InvocationEvent
PaintEvent

Dodatkowe zdarzenia zdefiniowane dla biblioteki SWING to:
(javax.swing.event.*):

AncestorEvent
CaretEvent
ChangeEvent
HyperlinkEvent
InternalFrameEvent
ListDataEvent
ListSelectionEvent
MenuDragMouseEvent
MenuEvent
MenuKeyEvent
PopupMenuEvent
TableColumnModelEvent
TableModelEvent
TreeExpansionEvent
TreeModelEvent
TreeSelectionEvent
UndoableEditEvent

Zdarzenia są powiązane z komponentami i tak:

Komponent	Zdarzenie
Adjustable	**AdjustmentEvent**
Applet	**ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent**
Button	**ActionEvent, FocusEvent, KeyEvent,**

Komponent	Zdarzenie
	MouseEvent, ComponentEvent
Canvas	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Checkbox	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
CheckboxMenuItem	ActionEvent, ItemEvent
Choice	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Component	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Container	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Dialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
FileDialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Frame	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Label	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
List	ActionEvent, FocusEvent, KeyEvent, MouseEvent, ItemEvent, ComponentEvent
Menu	ActionEvent
MenuItem	ActionEvent
Panel	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
PopupMenu	ActionEvent
Scrollbar	AdjustmentEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
ScrollPane	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextArea	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextComponent	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextField	ActionEvent, TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Window	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent

Każda klasa zdarzeń definiuje zbiór pól przechowujących stan zdarzenia. Stan zdarzenia jest istotny ze względu na obsługę zdarzenia, a więc na wykonanie odpowiedniego działania związanego z wartościami pól obiektu zdarzenia. Referencja do obiektu odpowiedniego zdarzenia jest podawana jako argument przy wykonywaniu metody obsługującej to zdarzenie. Metody związane ze zdarzeniami są pogrupowane jako zbiory abstrakcyjnych metod w interfejsach lub jako zbiory pustych metod w klasach (adaptery). Zadaniem programisty jest definicja metod wykorzystywanych do obsługi zdarzeń. Następujące interfejsy i adaptery są zdefiniowane w Javie:

Interfejs lub adapter	Metody
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)
TextListener	textValueChanged(TextEvent)

Wykonując metodę obsługującą zdarzenie można wykorzystać pola obiektu danego zdarzenia. Możliwość ta pozwala określić okoliczności wystąpienia zdarzenia np.: współrzędne kursora podczas generacji zdarzenia, typ klawisza klawiatury jaki był wciśnięty, nazwę komponentu źródłowego dla danego zdarzenia, stan lewego przycisku myszki podczas zdarzenia, itp. Analiza pól obiektu jest zatem niezwykle istotna z punktu widzenia działania tworzonego interfejsu graficznego.

Poniższy program ukazuje obsługę zdarzeń związanych z klawiaturą i myszką:

Przykład 5.15:

```
//Komunikator.java:


import java.awt.*;
import java.awt.event.*;

class Ekran extends Canvas{

        public String s="Witam";
        private Font f;

        Ekran (){
                super();
```

```
        f = new Font("Times New Roman",Font.BOLD,20);
        setFont(f);
        addKeyListener(new KeyAdapter(){
            public void keyPressed(KeyEvent ke){
                s=new String(ke.paramString());
                repaint();
            }
        });
        addMouseListener(new MouseAdapter(){
            public void mousePressed(MouseEvent me){
                s=new String(me.paramString());
                repaint();
            }
        });

    }

    public void paint(Graphics g){
        g.drawString(s,20,220);
    }


}// koniec class Ekran

public class Komunikator extends Frame  {


    Komunikator (String nazwa){
        super(nazwa);
    }

    public static void main(String args[]){
        Komunikator okno = new Komunikator("Komunikator");
        okno.setSize(600,500);
        Ekran e = new Ekran();
        okno.add(e);
        okno.addWindowListener(new WindowAdapter(){
             public void windowClosing(WindowEvent e){
                  System.out.println("Dziekujemy za prace z programem...");
                  System.exit(0);
             }
         });
        okno.setVisible(true);


    }
}// koniec public class Komunikator
```

W przykładzie tym zastosowano komponent Canvas stanowiący pole graficzne, w obrębie którego można rysować używając metod zdefiniowanych w klasie Graphics. Z polem graficznym związano obsługę dwóch typów zdarzeń: wciśnięcie klawisza klawiatury (keyPressed()) oraz wciśnięcie klawisza myszki (mousePressed()). Zdarzenia te są opisane w obiektach klas KeyEvent oraz MouseEvent. Każdy obiekt przechowuje informacje związane z powstałym zdarzeniem i dlatego można te pola

odczytać celem wyświetlenia ich wartości. W powyższym przykładzie zasotosowano metodę paramString() zwracającą wartości pól obiektu w postaci łańcucha znaków (String). W przypadku wystąpienia zdarzenia i jego obsługi za pomocą omawianych metod generowana jest metoda repaint() powodująca odświeżenie pola graficznego. Warto zauważyć, że w definiowaniu klas anonimowych obsługujących zdarzenia zastosowano adaptery. Jak wspomniano wcześniej jest to na tyle wygodne w przeciwieństwie od implementacji interfejsów, że nie trzeba definiować wszystkich metod w nich zawartych (zadeklarowanych).

Rozdział 6 Grafika i multimedia w Javie

6.1 Grafika (rysunki)
6.2 Czcionki
6.3 Kolor
6.4 Obrazy
6.5 Dźwięki
6.6 Java Media API

6.1 Grafika (rysunki)

Pakiet AWT zarówno w wersjach wcześniejszych jak i w wersji 2 wyposażony jest w klasę Graphics, a od wersji 2 dodatkowo w klasę Graphics2D. Klasy te zawierają liczne metody umożliwiające tworzenie i zarządzanie grafiką w Javie. Podstawą pracy z grafiką jest tzw. kontekst graficzny, który jako obiekt posiada właściwości konkretnego systemu prezentacji np. panelu. W AWT kontekst graficzny jest dostarczany do komponentu poprzez następujące metody:

- paint
- paintAll
- update
- print
- printAll
- getGraphics

Obiekt graficzny (kontekst) zawiera informacje o stanie grafiki potrzebne dla podstawowych operacji wykonywanych przez metody Javy. Zaliczyć tu należy następujące informacje:

- obiekt komponentu, który będzie obsługiwany,
- współrzędne obszaru rysowania oraz obcinania,
- aktualny kolor,
- aktualne czcionki,
- aktualna funkcja operacji na pikselach logicznych (XOR lub Paint),
- aktualny kolor dla operacji XOR.

Posiadając obiekt graficzny można wykonać szereg operacji rysowania np.: Graphics g;

g.drawLine(int x1, int y1, int x2, int y2) - rysuje linię pomiędzy współrzędnymi (x1,y1) a (x2,y2), używając aktualnego koloru,

g.drawRect(int x, int y, int width, int height) - rysuje prostokąt o wysokości height i szerokości width począwszy od punktu (x,y), używając aktualnego koloru,

g.drawString(String str, int x, int y) - rysuje tekst str począwszy od punktu (x,y), używając aktualnego koloru i czcionki,

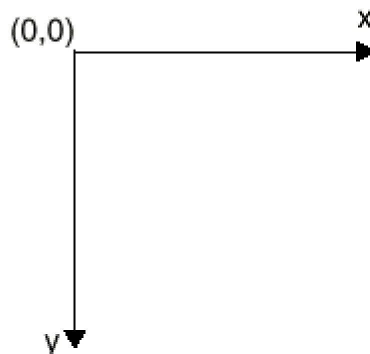
g.drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer) - wyświetla aktualnie dostępny zestaw pikseli obrazu img na tle o kolorze bgcolor,

począwszy od punktu (x,y)

g.setColor(Color c) - ustawia aktualny kolor c np. Color.red

g.setFont(Font font) - ustawia aktualny zestaw czcionek

W celu rysowania elementów grafiki konieczna jest znajomość układu współrzędnych w ramach, którego wyznacza się współrzędne rysowania. Podstawowym układem współrzędnych w Javie jest układ użytkownika, będący pewną abstrakcją układów współrzędnych dla wszystkich możliwych urządzeń. Układ użytkownika definiowany jest w sposób następujący.



Rysunek 6.1 Układ współrzędnych użytkownika

Układ współrzędnych konkretnego urządzenia może pokrywać się z układem użytkownika lub nie. W razie potrzeby współrzędne z układu użytkownika są automatycznie transformowane do właściwego układu współrzędnych danego urządzenia.
Jako ciekawostkę można podać, że JAVA nie definiuje wprost metody umożliwiającej rysowanie piksela, która w innych językach programowania służy często do tworzenia obrazów. W Javie nie jest to potrzebne. Do tych celów służą liczne metody drawImage(). Niemniej łatwo skonstruować metodę rysującą piksel np.

```
drawPixel(Color c, int x, int y){
   setColor(c);
   drawLine(x,y,x,y);
}
```

Pierwotne wersje AWT definiują kilka obiektów geometrii jak np. Point, Rectangle. Elementy te są bardzo przydatne dlatego, że, co jest właściwe dla języków obiektowych, nie definiujemy za każdym razem prostokąta za pomocą atrybutów opisowych (współrzędnych) lecz przez gotowy obiekt - prostokąt, dla którego znane są (różne metody) jego liczne właściwości.

Poniższa aplikacja umożliwia sprawdzenie działania prostych metod graficznych:

Przykład 6.1:

```
//Rysunki.java:

import java.awt.event.*;
```

```
import java.awt.*;

public class Rysunki extends Frame {
   Rysunki () {
      super ("Rysunki");
      setSize(200, 220);
   }
   public void paint (Graphics g) {
       Insets insets = getInsets();
       g.translate (insets.left, insets.top);
       g.drawLine (5, 5, 195, 5);
       g.drawLine (5, 75, 5, 75);

      g.drawRect (25, 10, 50, 75);
       g.fillRect (25, 110, 50, 75);
       g.drawRoundRect (100, 10, 50, 75, 60, 50);
       g.fillRoundRect (100, 110, 50, 75, 60, 50);

       g.setColor(Color.red);
       g.drawString ("Test grafiki",50, 100);
       g.setColor(Color.black);
   }
   public static void main (String [] args) {
      Frame f = new Rysunki ();
       f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
            System.out.println("Dziekujemy za prace z programem...");
            System.exit(0);
        }
      });
      f.setVisible(true);
   }
}// koniec public class Rysunki extends Frame
```

Wykorzystana w powyższym kodzie metoda translate() zmienia początek układu współrzędnych przesuwając go do aktywnego pola graficznego (bez ramek).

Java2D API w sposób znaczny rozszerza możliwości graficzne AWT. Po pierwsze umożliwia zarządzanie i rysowanie elementów graficznych o współrzędnych zmiennoprzecinkowych (float i double). Własność ta jest niezwykle przydatna dla różnych aplikacji m.in. dla aplikacji w medycynie (wymiarowanie, planowanie terapii, projektowanie implantów, itp.). Ta podstawowa zmiana podejścia do rysowania obiektów graficznych i geometrycznych powoduje powstanie, nowych, licznych klas i metod. W sposób szczególny należy wyróżnić tutaj sposób rysowania nowych elementów. Odbywa się to poprzez zastosowanie jednej metody:

```
Graphics2D g2;
g2.draw(Shape s);
```

Metoda draw umożliwia narysowanie dowolnego obiektu implementującego interfejs Shape (kształt). Przykładowo narysowanie linii o współrzędnych typu float można wykonać w następujący sposób:

```
Line2D linia = new Line2D.Float(20.0f, 10.0f, 100.0f, 10.0f);
g2.draw(linia);
```

Oczywiście klasa Line2D implementuje interfejs Shape. Java2D wprowadza liczne klasy w ramach pakietu java.awt.geom, np:

Arc2D.Double
Arc2D.Float
CubicCurve2D.Double
CubicCurve2D.Float
Dimension2D
Ellipse2D.Double
Ellipse2D.Float
GeneralPath
Line2D
Line2D.Double
Line2D.Float
Point2D
Point2D.Double
Point2D.Float
QuadCurve2D.Double
QuadCurve2D.Float
Rectangle2D
Rectangle2D.Double
Rectangle2D.Float
RoundRectangle2D.Double
RoundRectangle2D.Float

W celu skorzystania z tych oraz innych dobrodziejstw jakie wprowadza Java2D należy skonstruować obiekt graficzny typu Graphics2D. Ponieważ Graphics2D rozszerza klasę Graphics, to konstrukcja obiektu typu Graphics2D polega na:

```
Graphics2D g2 = (Graphics2D) g;
```

gdzie g jest obiektem graficznym otrzymywanym jak omówiono wyżej.

Uwaga! Argumentem metody paint komponentów jest obiekt klasy Graphics a nie Graphics2D.

Dodatkowe klasy w AWT wspomagające grafikę to BasicStroke oraz TexturePaint. Pierwsza z nich umożliwia stworzenie właściwości rysowanego obiektu takich jak np.: szerokość linii, typ linii. Przykładowo ustawienie szerokości linii na 12 punktów odbywać się może poprzez zastosowanie następującego kodu:

```
grubaLinia = new BasicStroke(12.0f);
g2.setStroke(grubaLinia);
```

Klasa TexturePoint umożliwia wypełnienie danego kształtu (Shape) określoną teksturą. Do dodatkowych zalet grafiki w Java2D należy zaliczyć:
- sterowanie jakością grafiki (np. antyaliasing, interpolacje)

- sterowanie przekształceniami geometrycznymi (przekształcenia sztywne - affiniczne - klasa AffineTransform),
- sterowanie przeźroczystością elementów graficznych,
- bogate narzędzia do zarządzania czcionkami i rysowania tekstu,
- narzędzia do drukowania grafiki,
- inne.

Przykładowa aplikacja ukazująca proste elementy grafiki w Java2D

Przykład 6.2:

```
//Rysunki2.java:

import java.awt.event.*;
import java.awt.geom.*;
import java.awt.*;

public class Rysunki2 extends Frame {
   Rysunki2 () {
      super ("Rysunki2");
      setSize(200, 220);
   }
   public void paint (Graphics g) {

        Graphics2D g2 = (Graphics2D) g;
        Insets insets = getInsets();
        g2.translate (insets.left, insets.top);

        Line2D linia = new Line2D.Float(20.0f, 20.0f, 180.0f, 20.0f);
        g2.draw(linia);

        BasicStroke grubaLinia = new BasicStroke(6.0f);
        g2.setStroke(grubaLinia);
        g2.setColor(Color.red);
        Line2D linia2 = new Line2D.Float(20.0f, 180.0f, 180.0f, 180.0f);
        g2.draw(linia2);
        g2.drawString ("Test grafiki",50, 100);
        g2.setColor(Color.black);
   }
   public static void main (String [] args) {
      Frame f = new Rysunki2 ();
      f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
             System.out.println("Dziekujemy za prace z programem...");
             System.exit(0);
        }
      });
      f.setVisible(true);
   }
} // koniec public class Rysunki2 extends Frame
```

6.2 Czcionki

W Javie można korzystać z różnych typów czcionek, które związane są z daną platformą na jakiej pracuje Maszyna Wirtualna. Dostęp do czcionek odbywa się poprzez trzy typy nazw: nazwy logiczne czcionek, nazwy czcionek, nazwy rodziny czcionek. Nazwy logiczne czcionek to nazwy zdefiniowane dla Javy. Możliwe są następujące nazwy logiczne czcionek w Javie: Dialog, DialogInput, Monospaced, Serif, SansSerif, oraz Symbol. Nazwy logiczne są odwzorowywane na nazwy czcionek powiązane z czcionkami dla danego systemu. Odwzorowanie to występuje w pliku font.properties znajdującego się w katalogu lib podkatalogu jre (Java Runtime Engine). W pliku tym zdefiniowano również sposób kodowania znaków. Z tego powodu w zależności od sposobu kodowania i typów wykorzystywanych czcionek definiowane są różne pliki font.properties z odpowiednim rozszerzeniem np. font.properties.pl. Przykładowe fragmenty plików opisujący właściwości czcionek to np.:

```
//Windows NT font.properties:
(...)
serif.0=Times New Roman,ANSI_CHARSET
serif.1=WingDings,SYMBOL_CHARSET,NEED_CONVERTED
(...)
```

```
//Solaris font.properties.pl:
(..)
serif.plain.0=-linotype-times-medium-r-normal--*-%d-*-*-p-*-iso8859-1
serif.1=-monotype-timesnewroman-regular-r-normal--*-%d-*-*-p-*-iso8859-2
(...)
```

Różnica pomiędzy nazwą logiczną czcionek w Javie a ich nazwami jest niewielka, np. nazwa logiczna Serif, nazwa serif. Różnice nazw ukazuje poniższy program:

Przykład 6.3:

```
//Nazwy.java

import java.awt.*;

public class Nazwy {

        private Font f;

        Nazwy (){

                f = new Font("Serif",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki Serif: "+f.getName());
                System.out.println("Oto nazwa  czcionki Serif: "+f.getFontName());
                f = new Font("SansSerif",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki SansSerif: " + f.getName());
                System.out.println("Oto nazwa  czcionki SansSerif: " +f.getFontName());
                f = new Font("Dialog",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki Dialog: "+f.getName());
                System.out.println("Oto nazwa  czcionki Dialog: "+f.getFontName());
```

```
                f = new Font("DialogInput",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki DialogInput: " + f.getName());
                System.out.println("Oto nazwa  czcionki DialogInput: "+ f.getFontName());
                f = new Font("Monospaced",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki Monospaced: "+ f.getName());
                System.out.println("Oto nazwa  czcionki Monospaced: "+ f.getFontName());
                f = new Font("Symbol",Font.PLAIN,12);
                System.out.println("Oto nazwa logiczna czcionki Symbol: "+ f.getName());
                System.out.println("Oto nazwa  czcionki Symbol: "+f.getFontName());
        }

        public static void main(String args[]){
                Nazwy czcionki = new Nazwy();

        }
}// koniec public class Nazwy
```

W pracy z Javą w celu stworzenia uniwersalnego kodu należy korzystać z nazw logicznych. Dla danej platformy można oczywiście używać nazw fontów tam zainstalowanych. W celu uzyskania informacji o zainstalowanych czcionkach na danej platformie należy posłużyć się metodami klasy GraphicsEnvironment (w poprzednich wersjach JDK należało korzystać z metody getFontList() klasy Toolkit). Poniższy program ukazuje zainstalowane czcionki w systemie poprzez podanie nazw rodzin czcionek oraz poszczególnych nazw.

Przykład 6.4:

```
//Czcionki.java:


import java.awt.*;
import java.awt.event.*;


public class Czcionki extends Frame  {

        public String s[];
        private Font czcionki[];
        private Font f;

        Czcionki (String nazwa){
                super(nazwa);
                f = new Font("Verdana",Font.BOLD,12);
                s=GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
                czcionki=GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts();


        }

        public static void main(String args[]){
                Czcionki okno = new Czcionki("Lista czcionek");
                okno.setSize(600,500);
                okno.setLayout(new GridLayout(2,1));
```

```
                List lista1 = new List(1, false);
                for (int i=0; i<okno.s.length; i++){
                        lista1.add(okno.s[i]);
                }
                lista1.setFont(okno.f);
                okno.add(lista1);

                List lista2 = new List(1, false);
                for (int i=0; i<okno.czcionki.length; i++){
                        lista2.add(okno.czcionki[i].toString());
                }
                lista2.setFont(okno.f);
                okno.add(lista2);

                okno.addWindowListener(new WindowAdapter(){
                        public void windowClosing(WindowEvent e){
                                System.out.println("Dziekujemy za prace z programem...");
                                System.exit(0);
                        }
                });
                okno.setVisible(true);


        }
}// koniec public class Czcionki
```

Pełny opis czcionek można uzyskać posługując się metodami zdefiniowanymi w klasie FontMetrics. Podstawowe metody umożliwiają między innymi uzyskanie informacji o szerokości znaku lub znaków dla wybranej czcionki (charWithd(), charsWidth()), o wysokości czcionki (getHight()), o odległości między linią bazową czcionki a wierzchołkiem znaku (getAscent()), o odległości między linią bazową czcionki a podstawą znaku (getDescent()), itd.
Ustawianie czcionek dla danego komponentu polega na stworzeniu obiektu klasy Font a następnie na wywołaniu metody komponentu setFont(), np.:

```
(...)
Component c;
(...)
Font f = new Font („Arial”, Font.PLAIN, 12);
c.setFont(f);
(...)
```

Wywołanie konstruktora klasy Font w powyższym przykładzie wymaga określenia nazwy lub nazwy logicznej czcionki, stylu, oraz rozmiaru czcionki. Styl czcionki może być określony przez następujące pola klasy Font:

Font.PLAIN – tekst zwykły,
Font.ITALIC – tekst pochyły (kursywa),
Font.BOLD – tekst pogrubiony,
Font.ITALIC + Font.BOLD – tekst pogrubiony i pochyły.

Do tej pory omówione zostały podstawowe własności grafiki dostarczanej przez Java2D z pominięciem obsługi obrazów i kolorów. Zarządzanie kolorami i metody definiowania, wyświetlania i przetwarzania obrazów są niezwykle istotne z punktu widzenia aplikacji medycznych. Dlatego teraz te zagadnienia zostaną osobno omówione.

6.3 Kolor

Kolor jest własnością postrzegania promieniowania przez człowieka. Średni obserwator (definiowany przez różne organizacje normalizacyjne) postrzega konkretne kolory jako wynik rejestracji promieniowania elektromagnetycznego o konkretnych długościach fal. Ponieważ system wzrokowy człowieka wykorzystuje trzy typy receptorów do rejestracji różnych zakresów promieniowania (będziemy dalej nazywać te zakresy zakresami postrzeganych kolorów) również systemy sztuczne tworzą podobne modele reprezentacji kolorów. Najbliższym spektralnie systemem wykorzystywanym dla celów reprezentacji kolorów jest system RGB (posiadający różne wady np. brak reprezentacji wszystkich barw, różnice kolorów często inne od tych postrzeganych przez człowieka (nieliniowa zależność postrzegania zmian intensywności), itp.). System RGB jest zależny od urządzenia, stąd istnieją różne jego wersje np. CIE RGB, PAL RGB, NTSC RGB, sRGB.

Standardowo przyjmuje się, że kolor na podstawie przestrzenni RGB definiowany jest jako liniowa kombinacja barw podstawowych:

kolor = r*R+g*G+b*B,

gdzie RBG to kolory (długości fal) podstawowe, a rgb to wartości stymulujące (tristimulus values) wprowadzające odpowiednie wagi z zakresu (0,1). Zakres wag w systemach komputerowych jest przeważnie modyfikowany do postaci zakresu dyskretnych wartości całkowitych o dynamice 8 bitowej czyli 0..255. Stąd też programista definiuje kolor jako zbiór trzech wag z zakresu 0..255, np., w Javie Color(0,0,255) daje barwę niebieską. W Javie istnieją predefiniowane stałe statyczne w klasie Color reprezentujące najczęściej wykorzystywane kolory np. Color.blue. Domyślną przestrzenią kolorów w Javie jest sRGB (wprowadzana jako standard dla WWW - http://www.w3.org/pub/WWW/Graphics/Color/sRGB.html), niemniej można korzystać z innych przestrzenni kolorów np. niezależnej sprzętowo CIE XYZ (CIE - Commission Internationale de L'Eclairage). Wprowadzono specjalny pakiet jawa.awt.color obsługujący przestrzenie kolorów oraz standardowe profile urządzeń bazując na specyfikacji ICC Profile Format Specification, Version 3.4, August 15, 1997, International Color Consortium. Podsumowując należy wskazać najbardziej istotne klasy w Javie związane z kolorem:
- java.awt.color.ColorSpace
- java.awt.Color
- java.awt.image.ColorModel

Warto zwrócić uwagę, że każda z wymienionych klas występuje w innym miejscu. Do tej pory omówiona została krótko rola klasy ColorSpace oraz Color. Ostatnia z prezentowanych klas ColorModel jest związana (jak nazwa pakietu sugeruje) z tworzeniem i wyświetlaniem obrazów. Klasa ColorModel opisuje bowiem konkretną metodę odwzorowania wartości piksela na dany kolor. W celu określenia koloru piksela zgodnie z przyjętą przestrzenią kolorów definiuje się w wybranym modelu komponenty kolorów i przeźroczystości (np. Red Green Blue Alpha). Metodę definiowania tych komponentów określa właśnie ColorModel.

Najbardziej znane formy zapisu komponentów to tablice komponentów lub jedna wartość 32 bitowa, gdzie każde 8 bitów wykorzystywane jest do przechowywania informacji o danych komponencie. Wykorzystywane podklasy klasy abstrakcyjnej ColorModel to: ComponentColorModel, IndexColorModel oraz PackedColorModel.

Pierwszy model - ComponentColorModel jest uniwersalną wersją przechowywania współrzędnych w wybranej przestrzeni kolorów, stąd nadaje się do reprezentacji dowolnej przestrzeni kolorów. Każda próbka (komponent) w tym modelu jest przechowywana oddzielnie. Dla każdego więc piksela tworzony jest zbiór odseparowanych próbek. Ilość próbek na piksel musi być taka sama jak ilość współrzędnych w przyjętej przestrzeni kolorów. Kolejność występowania próbek jest definiowana przez przestrzeń kolorów, przykładowo dla przestrzeni RGB - TYPE_RGB, index 0 oznacza próbkę dla komponentu RED, index 1 dla GREEN, index 2 dla BLUE. Typ danych dla przechowywania próbek może być 8-bitowy, 16-bitowy lub 32-bitowy: DataBuffer.TYPE_BYTE, DataBuffer.TYPE_USHORT, DataBuffer.TYPE_INT. Wywołując konstruktor tej klasy podaje się przede wszystkim przestrzeń kolorów, oraz tablicę o długości odpowiadającej liczbie komponentów w danej przestrzeni przechowującą dla każdego komponentu liczbę znaczących bitów.
Kolejny model - IndexColorModel - jest szczególnie wykorzystywany w aplikacjach medycznych, ponieważ odwołuje się on do koloru nie poprzez zbiór próbek lecz poprzez wskaźnik do tablicy kolorów. Definiując model kolorów zgodnie z tą klasą tworzymy tablicę kolorów, do której później odwołujemy się podając wskaźnik. Model taki jest wykorzystywany w różnych aplikacjach tam, gdzie tworzy się tak zwane palety kolorów, tablice kolorów (pseudokolorów). Jest to szczególnie ważne tam, gdzie posiadane wartości do tworzenia obrazu (wartości pikseli) nie reprezentują promieniowania widzialnego. Mówi się wówczas o tak zwanych sztucznych obrazach i tablicach pseudo-kolorów (np. obrazy w podczerwieni, obrazy ultradźwiękowe, obrazy Rtg, itp.). W medycynie prawie wszystkie metody obrazowania wykorzystują tablice pseudokolorów, a właściwie jedną tablicę - odcieni szarości (najczęściej R=G=B). Wywołując jeden z konstruktorów klasy IndexColorModel podaje się liczbę bitów na piksel, rozmiar tablicy oraz konkretne definicje kolorów poprzez zbiór trzech podstawowych próbek dla RGB.
Ostatni z przedstawianych modeli - PackedColorModel - jest abstrakcyjną klasą w ramach której kolor jest oznaczany dla danej przestrzeni kolorów poprzez definicję wszystkich komponentów oddzielnie, spakowanych w jednej liczbie 8, 16 lub 32 bitowej. Podklasą tej abstrakcyjnej klasy jest DirectColorModel. Przy wywołaniu konstruktora podaje się liczbę bitów na piksel oraz wartości masek dla każdego komponentu celem wskazania wartości tych komponentów. Obecnie model ten wykorzystuje się jedynie dla przestrzeni sRGB.

6.4 Obrazy

Możliwości pozyskiwania, tworzenia i przetwarzania obrazów w Javie są bardzo duże. Począwszy od pierwszych wersji w ramach AWT poprzez Java2D a skończywszy na tworzonej JAI (Java Advanced Imaging API - bardziej elastyczne definicje obrazów, bogate biblioteki narzędzi np. DFT) język JAVA dostarcza wiele narzędzi do tworzenia profesjonalnych systemów syntezy, przetwarzania, analizy i prezentacji obrazów.

W początkowych wersjach bibliotek graficznych Javy podstawą pracy z obrazami była klasa java.awt.Image. Obiekty tej abstrakcyjnej klasy nadrzędnej uzyskiwane są w sposób zależny od urządzeń. Podstawowa metoda zwracająca obiekt typu Image (klasa Image jest abstrakcyjna), często wykorzystywana w programach tworzonych w Javie to getImage(). Metoda ta dla aplikacji jest związana z klasą Toolkit, natomiast dla appletów z klasą Applet. Wywołanie metody getImage polega albo na podaniu ścieżki dostępu (jako String) lub lokalizatora URL do obrazu przechowywanego w formacie GIF lub JPEG. Przykładowo:

Image obraz = Toolkit.getDefaultToolkit.getImage("jacek.gif");
 - zwraca obiekt obraz na podstawie obrazu przechowywanego w pliku jacek.gif w bieżącej ścieżce dostępu

Image obraz = Toolkit.getDefaultToolkit.getImage(new URL("http://www-med.eti.pg.gda.pl/~jwr/icons/jwrs4.gif");
 - zwraca obiekt obraz na podstawie obrazu przechowywanego w pliku jwrs4.gif na serwerze www-med w danych podkatalogach.

Inną metodą uzyskania obiektu Image jest stworzenie obrazu poprzez wykorzystanie metody createImage(). Aby uzyskać obiekt typu Image za pomocą tej metody jako argument należy podać element implementujący interfejs ImageProducer. Obiekt będący wystąpieniem klasy implementującej ten interfejs jest odpowiedzialny za stworzenie (produkcję) obrazu jaki jest związany z obiektem typu Image. Przykładowo w poprzednich metodach getImage() obiekt typu Image jest często zwracany wcześniej niż stworzony zostanie (np. załadowany z sieci) obraz. Wówczas metody odwołujące się do obiektu Image zwrócą błąd. Dlatego obiekt typu ImageProducer informuje klientów (obserwatorów) o zakończonym procesie tworzenia obrazu związanego z obiektem typu Image. Klientów stanowią obiekty klas implementujących interfejs ImageObserver, których zachowanie jest ukazane poprzez jedyną metodę imageUpdate(). Metoda ta zgodnie z informacją od obiektu ImageProducer żąda odrysowania elementu (np. komponentu graficznego jak np. panel, applet, itd. - java.awt.Component implementuje interfejs ImageObserver). W czasie gdy ImageProducer wysyła informację o stanie do obiektu ImageObserver wysyła również dane do konsumenta czyli obiektu będącego wystąpieniem klasy implementującej interfejs ImageConsumer. Przykładowe klasy implementujące interfejs ImageConsumer to java.awt.image.ImageFilter oraz java.awt.image.PixelGrabber. Ponieważ do obiektu ImageConsumer dostarczane są wszystkie informacje związane z tworzonym obrazem (wymiary, piksele, model koloru) obiekt ten może za pomocą swoich metod wykonać wiele operacji na danych. Przykładowo obiekty klasy ImageFilter umożliwiają wycinanie próbek, filtrację kolorów, itp. natomiast obiekty klasy PixelGrabber umożliwiają pozyskać część lub wszystkie próbki z obrazu. Powracając do metody createImage(), której argumentem jest obiekt ImageProducer, umożliwia ona stworzenie (generację) obrazu na podstawie posiadanych danych. Konieczne jest jednak stworzenie obiektu typu ImageProducer, będącego argumentem tej metody. W tym celu wykorzystuje się klasę implementującą interfejs ImageProducer - MemoryImageSource. Klasa ta dostarcza szereg konstruktorów, którym podaje się: typ modelu kolorów (ColorModel) lub wykorzystuje się domyślny RGB, rozmiar tworzonego obrazu, wartości pikseli. Przykładowo w następujący sposób można wygenerować własny obiekt typu Image:

Przykład 6.5:

```
//Obraz.java:

import java.awt.event.*;
import java.awt.image.*;
import java.awt.*;

public class Obraz extends Frame {

Image ob;

  Obraz () {
     super ("Obraz");
     setSize(200, 220);
     ob=stworzObraz();
  }

  public Image stworzObraz(){
    int w = 100;  //szerokość obrazu
    int h = 100;   //wysokość obrazu
    int pix[] = new int[w * h]; //tablica wartości próbek
    int index = 0;
    //generacja przykładowego obrazu
    for (int y = 0; y < h; y++) {
      int red = (y * 255) / (h - 1);
      for (int x = 0; x < w; x++) {
        int blue = (x * 255) / (w - 1);
        pix[index++] = (255 << 24) | (red << 16) | blue;
      }
    }
    Image img = createImage(new MemoryImageSource(w, h, pix, 0, w));
    //tworzony jest obraz w RGB o szerokości w, wysokości h,
    //na podstawie tablicy próbek pix, bez przesunięcia w tej tablicy z w elementami w linii
    return img;
  }

  public void paint (Graphics g) {
    Insets insets = getInsets();
    g.translate (insets.left, insets.top);
    g.drawImage(ob,50,50,this);
  }

  public static void main (String [] args) {
    Frame f = new Obraz ();
     f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.out.println("Dziekujemy za prace z programem...");
          System.exit(0);
        }
    });
    f.setVisible(true);
  }
}//koniec public class Obraz extends Frame
```

Mając obiekt typu Image można obraz z nim związany wyświetlić (narysować). W tym celu należy wykorzystać jedną z metod drawImage(). W najprostszej metodzie drawImage() podając jako argument obiekt typu Image, współrzędne x,y oraz obserwatora (ImageObserver, często w ciele klasy komponentu, w którym się rysuje odwołanie do obserwatora jest wskazaniem aktualnego obiektu komponentu - this) możemy wyświetlić obrazu w dozwolonym do tego elemencie. Inna wersja metody drawImage() umożliwia skalowanie wyświetlanego obrazu poprzez podanie dodatkowych argumentów: szerokość (width) i wysokość (height).
Oprócz poznanych do tej pory operacji związanych z obrazami (stworzenie obiektu Image, filtracja danych, wyświetlenie i skalowanie) często wykorzystuje się dwie z kilku istniejących metod klasy Image, a mianowicie Image.getWidth() oraz Image.getHeight(). Metody te umożliwiają poznanie wymiarów obrazu, co jest często niezwykle istotne z punktu widzenia ich wyświetlania i przetwarzania. Argumentem obu metod jest ImageObserver (powiadomienie o dostępnych danych) po to aby można było uzyskać dane o stworzonym obrazie dla istniejącego już obiektu Image.

Opisane dotąd metody pracy z obrazami są historycznie pierwsze, a ich znaczne rozszerzenie daje Java2D. Java2D API rozszerza a zarazem zmienia koncepcję pracy z obrazami w Javie. Podstawową klasą jest w tej koncepcji klasa BufferedImage, będącą rozszerzeniem klasy Image z dostępnym buforem danych. Obiekt BufferedImage może być stworzony bezpośrednio w pamięci i użyty do przechowywania i przetwarzania danych obrazu uzyskanego z pliku lub poprzez URL.
Obraz BufferedImage może być wyświetlony poprzez użycie obiektów klasy Graphics2D. Obiekt BufferedImage zawiera dwa istotne obiekty: obiekt danych - Raster oraz model kolorów ColorModel. Klasa Raster umożliwia zarządzanie danymi obrazu. Na obiekt tej klasy składają się obiekty DataBuffer oraz SampleModel. DataBuffer stanowi macierz wartości próbek obrazu, natomiast SampleModel określa sposób interpretacji tych próbek. Przykładowo dla danego piksela próbki (RGB) mogą być przechowywane w trzech różnych macierzach (banded interleved) lub w jednej macierzy w formie przeplatanych próbek (pixel interleved) dla różnych komponentów (R1G1B1R2G2B2...). Rolą SampleModel jest określenie jakiej formy użyto do zapisu danych w macierzach. Najczęściej nie tworzy się bezpośrednio obiektu Raster lecz wykorzystuje się efekt działania obiektu BufferedImage, który rozbija Image na Raster oraz ColorModel. Niemniej istnieje możliwość stworzenia obiektu Raster poprzez stworzenie obiektu WritableRaster i podaniu go jako argumentu w jednym z konstruktorów klasy BufferedImage. Najbardziej popularne rozwiązanie tworzące obiekt klasy BufferedImage przedstawia następujący przykład:

```
URL url = ...

Image img = getToolkit().getImage(url);
 try {
    //pętla w której czekamy na skonczenie produkcji obrazu dla obiektu img
    MediaTracker mt = new MediaTracker(this);
    mt.addImage(img, 0);
    mt.waitForID(0);
 } catch (Exception e) {}
 int iw = img.getWidth(this);
 int ih = img.getHeight(this);
```

```
BufferedImage bi = new BufferedImage(iw, ih, BufferedImage.TYPE_INT_RGB);
//tworzymy obiekt BufferedImage
Graphics2D g2 = bi.createGraphics(); //określamy kontekst graficzny dla obiektu
g2.drawImage(img, 0, 0, this);
//wrysowujemy obraz do bufora - wypełniamy DataBuffer obiektu BufferedImage
```

Dla tak stworzonego obiektu BufferedImage można następnie przeprowadzić liczne korekcje graficzne, dodać elementy graficzne, zastosować metody przetwarzania obrazu oraz wyświetlić obraz. Wyświetlenie obrazu obiektu BufferedImage odbywa się poprzez wykorzystanie metod drawImage() zdefiniowanych dla klasy Graphics2D (np. g2.drawImage(bi,null,null);). Korekcje graficzne i dodawanie elementów graficznych polega na rysowaniu w stworzonym kontekście graficznym dla obiektu BufferedImage. Przetwarzanie obrazu odbywa się głównie poprzez wykorzystanie klas implementujących interfejs BufferedImageOp a mianowicie: AffineTransformOp, BandCombineOp, ColorConvertOp, ConvolveOp, LookupOp, oraz RescaleOp.

Do najczęściej wykorzystywanych operacji należą przekształcenia geometryczne sztywne (affiniczne) - aplikacja AffineTransformOp - oraz operacje splotu. Te ostatnie wykorzystuje się do tworzenia filtrów cyfrowych, za pomocą których można zmieniać jakość obrazu oraz wykrywać elementy obrazu jak np. linie, punkty, itp. Poniższy przykład ukazuje możliwość implementacji filtru cyfrowego oraz przykładową operację skalowania.

```
float[] SHARPEN3x3_3 = { 0.f, -1.f,  0.f,            //maska filtru cyfrowego
                        -1.f,  5.f, -1.f,
                         0.f, -1.f,  0.f};
BufferedImage bi=...

AffineTransform at = new AffineTransform();
at.scale(2.0, 2.0);
//określenie operacji geometrycznej - skalowanie wsp. 2
BufferedImageOp biop = null;
BufferedImage bimg = new BufferedImage (w,h,BufferedImage. TYPE_INT_RGB);
                                              // tworzymy nowy bufor
Kernel kernel = new Kernel(3,3,SHARPEN3x3_3);  //tworzymy kernel - jądro splotu
ConvolveOp cop = new ConvolveOp(kernel, ConvolveOp.EDGE_NO_OP, null);
  //definiujemy operację splotu z przepisywaniem wartości krawędzi
cop.filter(bi,bimg); //wykonujemy operację splotu
biop = new AffineTransformOp(at, AffineTransformOp.TYPE_NEAREST_NEIGHBOR);
   //definiujemy operację skalowania i interpolacji obrazu
g2.drawImage(bimg,biop,x,y); //rysujemy skalowany obraz
```

Poniższy przykład ukazuje możliwość implementacji filtru dolnoprzepustowego:

Przykład 6.6:

```
//Obraz2.java:

import java.awt.event.*;
import java.awt.image.*;
```

```
import java.awt.*;

public class Obraz2 extends Frame {

BufferedImage bi;
boolean stan=false;
Button b1;

   Obraz2 () {
      super ("Obraz2");
      setSize(200, 220);
      gUI();
      bi=stworzObraz();
   }

   public void gUI(){
       setLayout(new BorderLayout());
       Panel kontrola = new Panel();
       b1 = new Button("Filtracja");
       b1.addActionListener(new B1());
       kontrola.add(b1);
       add(kontrola,BorderLayout.SOUTH);
   }

   public BufferedImage stworzObraz(){

       int w = 100;  //szerokość obrazu
      int h = 100;   //wysokość obrazu
      int pix[] = new int[w * h]; //tablica wartości próbek
      int index = 0;
      for (int y = 0; y < h; y++) {
         int red = (y * 255) / (h - 1);
         for (int x = 0; x < w; x++) {
            int blue = (x * 255) / (w - 1);
            pix[index++] = (255 << 24) | (red << 16) | blue;
         }
      }
       Image img = createImage(new MemoryImageSource(w, h, pix, 0, w));
          BufferedImage bimg = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
       Graphics2D g2 = bimg.createGraphics();
       g2.drawImage(img, 0, 0, this);
       return bimg;
   }


   public void paint (Graphics g) {
      Insets insets = getInsets();
     g.translate (insets.left, insets.top);
     Graphics2D g2d = (Graphics2D) g;
     g2d.drawImage(bi,null,50,50);
   }

   class B1 implements ActionListener {
       public void actionPerformed(ActionEvent e) {
          float lp1=1.0f/9.0f;
```

```
            float l=4.0f*lp1;
            float[] lp = { lp1, lp1,  lp1,
                           lp1,  l, lp1,
                           lp1, lp1,  lp1};

            if (!stan){
              Kernel kernel = new Kernel(3,3,lp);
              ConvolveOp cop = new ConvolveOp(kernel, ConvolveOp.EDGE_NO_OP, null);
               bi = cop.filter(bi,null);
               stan=true;
               b1.setLabel("Powrot");
            }else
            {
               bi=stworzObraz();
               stan=false;
               b1.setLabel("Filtracja");
            }
            validate();
            repaint();
     }
    }

    public static void main (String [] args) {
        Frame f = new Obraz2 ();
         f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
               System.out.println("Dziekujemy za prace z programem...");
                System.exit(0);
            }
        });
        f.setVisible(true);
    }
}// koniec  public class Obraz2 extends Frame
```

6.5 Dźwięki

Do wersji JDK 1.2 można korzystać z obsługi plików dźwiękowych w formacie au (8-bit, mono), tylko w obrębie apletu. W celu odtworzenia dźwięku zapisanego w pliku o formacie au należy wykorzystać jedną z dwóch metod zdefiniowaną w klasie Applet:

public void play(URL url); gdzie „url” to adres pliku wraz z nazwą pliku,
lub
public void play(URL url, String nazwa); gdzie „url” to adres pliku, a „nazwa” to nazwa pliku.

Przykładowo można zastosować następujący kod:

Przykład 6.7:

```
//Graj.java:
```

```
import java.applet.*;

public class Graj extends Applet{

        public void init(){
                String nazwa = getParameter(„muzyka”);
                if(nazwa != null) play(getDocumentBase(),nazwa);
        }

}// koniec public class Graj

//--------------------------------------------
//Graj.html:

(...)
<param name=muzyka value=witam.au>
(...)
```

Powyższy przykład rozpocznie odtwarzanie pliku dźwiękowego o nazwie pobranej z pola parametru w pliku html, przechowywanego w tym samym katalogu na serwerze co dokument html. Ponieważ kod zawarto w metodzie init() odtwarzanie dźwięku będzie jednorazowe. W celu odtwarzania dźwięku w pętli można wykorzystać interfejs AudioClip. Zdefiniowane w klasie Applet metody getAudioClip() wywoływane tak samo jak metody play(), zwracają obiekt typu AudioClip (nie jawna definicja metod interfejsu). Dostępne metody umożliwiają zarówno jednorazowe odtworzenie dźwięku (play()) jak i odtwarzanie w pętli loop(()) oraz zatrzymanie odtwarzania dźwięku (stop()).

6.6 Java Media API

Pracę z wieloma mediami w Javie w znaczny sposób rozszerzają dodatkowe bibliteki Javy (Extensions) oznaczane przez javax. W ramach projektu Java Media API stworzono różne biblioteki ułatwiające pracę z mediami. Do podstawowych bibliotek należy zaliczyć m.in:

Java 2D – dwu-wymiarowa grafika i obrazowanie
Java 3D – trój-wymiarowa grafika i obrazowanie
Java Advanced Imaging – rozszerzenie Java2D pod względem reprezentacji i przetwarzania obrazów
Java Media Framework – przechwytywanie i odtwarzanie mediów (audio-wideo) ,
Java Speech – synteza i rozpoznawanie dźwięku,
Java Sound – przechwytywanie i odtwarzanie dźwięków.

W wersji JDK 1.3 dostępna jest standardowo biblioteka Java Sound, umożliwiająca nagrywanie i odtwarzanie dźwięków w różnych formatach.

Rozdział 7 Strumienie, operacje wejścia-wyjścia

7.1 Strumienie
7.2 Standardowe obsługa wejścia-wyjścia - klasy InputStream oraz OutputStream
  7.2.1 Obsługa wejścia – klasa InputStream
  7.2.2 Obsługa wejścia – klasa OutputStream
  7.2.3 Obsługa plików
7.3 Obsługa strumieni tekstu
7.4 Dzielenie strumienia – klasa StreamTokenizer
7.5 Strumienie poza java.io

7.1 Strumienie

W celu stworzenia uniwersalnego modelu przepływu danych wykorzystuje się w Javie model strumieni danych. Strumień (*stream*) jest sekwencją danych, zwykle bajtów. Pochodzenie i typ sekwencji danych zależny jest od danego środowiska. Podstawowe typu strumieni to te związane z operacjami wprowadzania danych do programu (operacje wejścia) i z operacjami wyprowadzania danych poza program (operacje wyjścia). W Javie do obsługi operacji wejścia stworzono klasę *InputStream*, natomiast dla obsługi operacji wyjścia stworzono klasę *OutputStream*. Strumień związany jest również z typem obszaru (urządzenia) z/do którego sekwencja danych przepływa oraz typem danych. Podstawowe obszary (urządzenia) to np. pamięć operacyjna, dyski (pliki), sieć, drukarka, ekran, itp. Typy danych jakie mogą być wykorzystywane przy przesyłaniu danych to np. *byte, String, Object*, itp. Zgodnie z dystrybucją JDK 1.0 klasy *OutputStream* oraz *InputStream* reprezentują strumienie jako sekwencje bajtów, czyli elementów typu *byte*. Często jednak zachodzi potrzeba formatowania danych strumienia. Można to wykonać w Javie korzystając z różnych klas formatujących dziedziczących z *OutputStream* i *InputStream*. Dobrym przykładem jest obsługa danych tekstowych wyświetlanych na standardowym urządzeniu wyjścia. Zgodnie z tym co było omawiane wcześniej klasyczną metodą wydruku tekstu w konsoli jest użycie polecenia *System.out.println();* gdzie *out* jest obiektem klasy *PrintStream*, stanowiącej klasę formatującą bajty sekwencji pochodzących z *OutputStream* na tekst. Rozpatrując dalej różnorodność obsługi strumieni w Javie należy wspomnieć o dodatkowych klasach wprowadzonych z wersją JDK 1.1, a mianowicie klasach *Reader* oraz *Writer*. Klasy te stanowią analogię do klas *InputStream* oraz *OutputStream*, niemniej przygotowane są do obsługi danych tekstowych (*String*). W omawianych wcześniej rozdziałach użyto już przykładowej klasy dziedziczącej z klasy *Writer*, a mianowicie klasy *PrintWriter* (zamiast *PrintStream*). Wprowadzenie dodatkowych klas obsługujących sekwencje łańcuchów znaków miało na celu ujednolicenie pracy w środowisku Javy z tekstem zapisywanym kodowanym w Unicodzie (2 bajty na znak). Dodatkowe, oddzielne klasy strumieni to klasa *StreamTokenizer* dzieląca strumień tekstowy na leksemy oraz klasa *RandomAccessFile* obsługująca pliki zawierające rekordy o znanych rozmiarach, tak że można dowolnie poruszać się w obrębie rekordów i je modyfikować. Ważnym zagadnieniem związanym ze strumieniami jest możliwość zapisu obiektu jako sekwencji bajtów i przesłanie go do programu (metody) działającej zdalnie. Efekt ten jest uzyskiwany poprzez zastosowanie mechanizmu serializacji i wykorzystania klas *ObjectInputStream* oraz *ObjectOutputStream*.

Wszystkie omawiane klasy obsługujące różne typy strumieni są zdefiniowane w pakiecie java.io.
Wykorzystanie strumieni jest powszechne w tworzeniu programów tak więc warto bliżej zapoznać się z podstawowymi klasami je obsługującymi.

7.2 Standardowe obsługa wejścia-wyjścia - klasy InputStream oraz OutputStream

7.2.1 Obsługa wejścia – klasa InputStream

Klasa ta jest klasą abstrakcyjną i zawiera podstawowe metody umożliwiające odczytywanie i kontrolę bajtów ze strumienia. Ponieważ jest to klasa abstrakcyjna nie tworzy się dynamicznie obiektu tej klasy, lecz można go uzyskać poprzez odwołanie się do standardowego wejścia zainicjowanego zawsze w polu *in* klasy *System*, czyli *System.in*. Inne możliwości uzyskania obiektu klasy *InputStream* to wywołanie metod zwracających referencję do obiektu tego typu np.: metoda *getInputStream*() zdefiniowana w klasie *Socket*. Jedyną metodą abstrakcyjną (czyniącą z tej klasy klasę abstrakcyjną) jest metoda *read*() oznaczająca czytanie kolejnego bajtu ze strumienia wejścia. Pozostałe metody umożliwiają:
- odczyt bajtów do zdefiniowanej tablicy:

```
int read(byte b[]);
int read(byte b[], int offset, int length);
```

- pominięcie określonej liczby bajtów w odczycie:

```
long skip(long n);
```

- kontrolę stanu strumienia (czy są dane):

```
int available();
```

- tworzenie markerów:

```
boolean markSupported(); kontrola czy tworzenie markerów jest możliwe
synchronized void mark(int readlimit);
synchronized void reset()
```

- zamknięcie strumienia

```
void close();
```

Prawie wszystkie metody (poza *markSupported*() oraz *mark*()) mogą wygenerować wyjątek, który musi być zadeklarowany lub obsługiwany w kodzie programu. Podstawową klasą wyjątków jest tutaj klasa *IOException*.
Pokazane wyżej metody *read*() blokują dostęp tak długo, jak dane są dostępne lub wystąpi koniec pliku albo wygenerowany zostanie wyjątek.
Klasy dziedziczące z klasy *InputStream* to:

- ByteArrayInputStream - definiuje pole bufora bajtów,
- FileInputStream - umożliwia odczyt pliku (strumień z pliku),
- FilterInputStream - praktycznie kopiuje funkcjonalność InputStream celem jej wykorzystania w klasach dziedziczących z FilterInputStream wprowadzającą dodatkowe narzędzia do obsługi sekwencji bajtów:

BufferedInputStream, CheckedInputStream, DataInputStream, DigestInputStream, InflaterInputStream, LineNumberInputStream, ProgressMonitorInputStream, PushbackInputStream

np. DataInputStream umożliwia programowi odczyt danych zgodnie z podstawowymi formatami zdefiniowanymi w Javie (char, int, long, double, itp.).

- ObjectInputStream - dokonuje rekonstrukcji obiektu z sekwencji bajtów (stworzonej w wyniku serializacji i zapisu metodą writeObject() klasy ObjectOutputStream),
- PipedInputStream - dokonuje przepływu danych do odpowiadającemu obiektowi klasy PipedOutputStream, który jest podawany jako argument konstruktora klasy PipedInputStream,
- SequenceInputStream, dokonuje logicznej koncentracji strumieni w jeden,
- StringBufferInputStream - tworzy strumień z podanego łańcucha znaków (obiektu String) - klasa ta jest oznaczona w JDK 1.2 jako deprecated ponieważ nie dokonuje właściwej konwersji znaków na bajty. Postuluje się jej zastąpienie klasą StringReader.

7.2.2 Obsługa wyjścia – klasa OutputStream

W podobny sposób, niemniej dotyczący obsługi wyjścia, definiowane są klasy dziedziczące z klasy *OutputStream*. Klasa ta jest również klasą abstrakcyjną z jedyną abstrakcyjną metodą *write*() zapisująca kolejny bajt do strumienia. Podstawowe metody tej klasy to:

void close() - zamknięcie strumienia,
void flush() - przesuwa buforowane dane do strumienia,
void write(byte[] b, int off, int len),
void write(byte[] b) - zapisują dane z tablicy b do strumienia wyjścia.

Klasy dziedziczące z klasy OutputStream to:

- ByteArrayOutputStream,
- FileOutputStream,
- FilterOutputStream,
- ObjectOutputStream,
- PipedOutputStream.

Klasy te obsługują strumień wyjściowy a ich funkcjonalność jest analogiczna do omawianych wyżej wersji obsługujących wejście.

Przykład 7.1:

```
//Echo.java

import java.io.*;
```

```
public class Echo{

	public static void main(String args[]){
		byte b[] = new byte[100];
		try{
			System.in.read(b);
			System.out.write(b);
			System.out.flush();
		} catch (IOException ioe){
			System.out.println("Błąd wejścia-wyjścia");
		}

	}

}//koniec public class Echo
```

Powyższy program ukazuje proste zastosowanie strumieni. Początkowo wykorzystywany jest istniejący strumień wejścia *System.in* w celu wczytania danych ze standardowego urządzenia wejścia (klawiatura). Dane są wczytywane aż wciśnięty zostanie klawisz *Enter* (koniec danych). Wczytane znaki są zapisywane do bufora b, który to jest następnie wykorzystywany do pobrania danych celem wysłania ich do strumienia wyjścia. Przyjętym w powyższym przykładzie strumień wyjścia to *System.out*. Warto zauważyć, że wykorzystano metody *write*() oraz *flush*() do zapisu danych. W efekcie działania programu otrzymujemy echo (powtórzenie) napisu wprowadzonego z klawiatury.

Kolejny przykład ukazuje możliwość dostępu do pliku.

Przykład 7.2:

```
//ZapiszPlany.java:

import java.io.*;

class Plany implements Serializable{

	private int liczbaLegionow;
	private int liczbaDzial;
	private String haslo;

	public void ustaw(int IL, int ID, String h){
		this.liczbaLegionow=IL;
		this.liczbaDzial=ID;
		this.haslo=h;
	}
	public void wyswietl(){
		System.out.println("Liczba legionów = "+liczbaLegionow);
		System.out.println("Liczba dział = "+liczbaDzial);
		System.out.println("Hasło dostępu = "+haslo);
	}
}// koniec class Plany

class Nadawca extends Thread{
```

```
	private String plikDanych;
	ZapiszPlany zp;
	Nadawca(String s, ZapiszPlany zp){
		this.plikDanych=s;
		this.zp=zp;
		setName("Nadawca");
	}
	public void run(){
		byte b[] = new byte[100];
		try{
			System.out.println("Podaj hasło");
			System.in.read(b);
			String s= new String(b);
			System.out.println("Zapisuję do pliku");
			Plany p = new Plany();
			p.ustaw(1,2,s);
			ObjectOutputStream o = new ObjectOutputStream(new
FileOutputStream(plikDanych));
			o.writeObject(p);
			o.flush();
			o.close();
		} catch (IOException ioe){
			System.out.println("Błąd wejścia-wyjścia");
		}
		zp.ustaw();
	}

}// koniec class Nadawca

class Odbiorca extends Thread{
	private String plikDanych;
	ZapiszPlany zp;
	Odbiorca(String s, ZapiszPlany zp){
		this.plikDanych=s;
		this.zp=zp;
		setName("Odbiorca");

	}
	public void run(){
		while( (!(this.isInterrupted())) && (zp.pobierz()) ){

		}
		try{
			System.out.println("Odczyt");
			ObjectInputStream i = new ObjectInputStream(new FileInputStream(plikDanych));
			Plany p = (Plany) i.readObject();
			p.wyswietl();
			i.close();
		} catch (Exception e){
			System.out.println("Błąd wejścia-wyjścia lub brak klasy Plany");
		}


	}
```

```
}// koniec class Odbiorca


public class ZapiszPlany{
        static boolean czekaj=true;
        synchronized void ustaw(){
                czekaj=false;
        }
        synchronized boolean pobierz(){
                return czekaj;
        }
        public static void main (String args[]){
                ZapiszPlany z = new ZapiszPlany();
                Nadawca n = new Nadawca("plany.txt",z);
                Odbiorca o = new Odbiorca("plany.txt",z);
                o.start();
                n.start();

        }

}//koniec public class ZapiszPlany
```

Powyższy program demonstruje zastosowanie obsługi strumieni dostępu do plików oraz wykorzystanie serializacji obiektów. Stworzono w kodzie cztery klasy. Pierwsza zawiera definicje zbioru pól (dane) oraz metod dostępu do nich. Klasa implementuje interfejs *Serializable* w celu umożliwienia zapisu obiektu do strumienia. Kolejne dwie klasy opisują zachowanie wątków generującego zapis danych i odczytującego zapis danych. W klasie Nadawca zawarto kod wczytujący zestaw znaków w klawiatury, który jest przypisywany do pola obiektu klasy Plany. Następnie tworzony jest strumień dostępu do pliku o podanej nazwie sformatowany do przesyłania obiektów. Zapis do bufora i przesłanie do strumienia odbywa się poprzez metody *writeObject*() oraz *flush*(). Po zakończeniu procesu zapisu danych do pliku powiadamiany jest drugi wątek, opisany w klasie Odbiorca. Zapisany obiekt jest odczytywany poprzez metodę *readObject*() a następnie jest wykonywana jedna z metod obiektu wyświetlająca stan danych. Jak widać przesłanie przez strumień obiektów jest ciekawym rozwiązaniem i umożliwia tworzenie rozproszonych aplikacji.

7.2.3 Obsługa plików

Dostęp do plików zaprezentowany wcześniej wykorzystywał klasy *FileInputStream* i *FileOutputStream*. Konstruktory tych klas umożliwiają inicjacje strumienia poprzez podanie jako argumentu albo nazwy pliku poprzez obiekt typu *String* lub poprzez podanie nazwy logicznej reprezentowanej przez obiekt klasy *File*. Klasa *File* opisuje abstrakcyjną reprezentację ścieżek dostępu do plików i katalogów. Ścieżka dostępu do pliku może być sklasyfikowana ze względu na jej zasięg lub ze względu na środowisko dla którego jest zdefiniowana. W pierwszym przypadku dzieli się ścieżki dostępu na absolutne i relatywne. Absolutne to te, które podają adres do pliku względem głównego korzenia systemu plików danego środowiska. Relatywne to te, które adresują plik względem katalogu bieżącego. Druga klasyfikacja rozróżnia ścieżki dostępu pod względem środowiska dla którego jest ona zdefiniowana, co w praktyce dzieli ścieżki dostępu na te zdefiniowane dla systemów opartych na UNIX i

na te zdefiniowane dla systemów opartych na MS *Windows* (własność systemu o nazwie file.separator). Przykłady:

a.) absolutna ścieżka dostępu:
UNIX: /utl/software/java/projekty
MS Windows: c:\utl\softare\java\projekty

b.) relatywna ścieżka dostępu:
UNIX: java/projekty
MS Windows: java\projekty.

Tworząc obiekt klasy *File* dokonywana jest konwersja łańcucha znaków na abstrakcyjną ścieżkę dostępu do pliku (abstrakcyjna ścieżka dostępu do pliku jest tworzona według określonych reguł podanych w dokumentacji API). Metody klasy *File* umożliwiają liczne kontrolę podanej ścieżki i plików (np. *isFile(), isDirectory(), isHidden, canRead(),* itp.) oraz dokonywania konwersji (np. *getPath(), getParent(), getName(), toURL(),* itp.) i wykonywania prostych operacji (*list() mkdir(),* itp.).

Uwaga! Należy pamiętać, że zapis tekstowy ścieżki dostępu dla środowiska MS Windows musi zawierać podwójny separator, gdyż pojedynczy znak umieszczony w inicjacji łańcucha znaków oznacz początek kodu ucieczki, np. „c:\\java\\kurs\\wyklad\\np".

Przykład 7.3:

```
// PobierzDane.java:

import java.io.*;

public class PobierzDane{

	public static void main(String args[]){
		File f = new File("DANE1");
		if (f.mkdir()) {
			File g = new File (".");
			String s[] = g.list();
			for (int i =0; i<s.length; i++){
				System.out.println(s[i]);
			}
		} else {
			System.out.println("Błąd operacji I/O");
		}

	}


}//koniec public class PobierzDane
```

Program PobierzDane ukazuje ciekawą własność obiektu *File*. Otóż <u>stworzenie obiektu tej klasy nie oznacza otwarcia strumienia czy stworzenia uchwytu do pliku</u>. Obiekt klasy *File* może więc być stworzony praktycznie dla dowolnej nazwy ścieżki. W prezentowanym przykładzie utworzono katalog o nazwie „DANE1", a następnie

dokonano wydruku plików i katalogów zawartych pod aktualnym adresem ścieżki (pod tym, z którego wywołano program **java**).

Pracę z plikami o swobodnym dostępie ułatwia zastosowanie innej klasy obsługującej operacje wejścia-wyjścia, a mianowicie klasy *RandomAccessFile*. Zdefiniowana jest w klasie obsługa plików zawierających rekordy o znanych rozmiarach, tak że można dowolnie poruszać się w obrębie rekordów i je modyfikować. Dane w pliku są interpretowane jako dane w macierzy do której dostęp jest możliwy poprzez odpowiednie ustawienie głowicy czytającej czy zapisującej dane. Zdefiniowano w tej klasie metody przesuwania głowicy (*getFilePointer*(), *seek*()) oraz szereg metod czytania i zapisu różnych typów danych.

7.3 Obsługa strumieni tekstu

W związku z problemem wynikającym z konwersji znaków Javy (Unicode) na bajty i odwrotnie występujących we wczesnych (JDK 1.0) realizacjach klas obsługi strumieni począwszy od wersji JDK1.1 wprowadzono dodatkowe klasy *Reader* i *Writer*. Obie abstrakcyjne klasy są analogicznie skonstruowane (dziedziczenie z klasy *Object* i deklaracja metod) jak klasy *InputStream* oraz *OutputStream*. Dziedziczące z nich klasy umożliwiają prostą i formatowaną obsługę sekwencji tekstu:

Reader:
- BufferedReader – buforuje otrzymywany tekst,
  - LineNumberReader – przechowuje dodatkowo informacje o numerze linii
- CharArrayReader – wprowadza bufor znaków do odczytu,
- FilterReader – klasa abstrakcyjna formatowania danych tekstowych,
  - PushbackReader – przygotowuje dane odesłania do strumienia,
- InputStreamReader – czyta bajty zamieniające je na tekst według podanego systemu kodowania znaków,
  - FileReader – odczyt danych tekstowych z pliku dla domyślnego systemu kodowania znaków, poprzez podanie ścieżki zależnej systemowo (String) lub abstrakcyjnej (File)
- PipedReader – obsługa potoku (związanie z klasą odczytującą),
- StringReader – obsługa strumienia pochodzącego od obiektu klasy String.

Konstrukcja klasy Writer jest analogiczna do Reader, z tym, że definiowany jest zapis zamiast odczytu. Podobnie wygląda struktura klas dziedziczących z Writer:

Writer:
- BufferedWriter,
- CharArrayWriter,
- FilterWriter,
- OutputStreamWriter,
  - FileWriter
- PipedWriter,
- PrintWriter, - formatowanie danych do postaci tekstu (analogiczna do PrintStream)

StringWriter

Odczyt danych odbywa się poprzez zastosowanie metod *read*() lub *readLine*() natomiast zapis danych do strumienia poprzez wykorzystanie metod write(). Zastosowanie klas dziedziczących z *Reader* i *Writer* ma dwie zasadnicze zalety: właściwa konwersja bajtów na znaki w Unicodzie i odwrotnie oraz możliwość zdefiniowania systemu kodowania znaków. W celu zobrazowania sposobu korzystania z tych klas warto zapoznać się z następującym przykładem:

Przykład 7.4:

```
//Czytaj.java:

import java.io.*;

/*Plik „plik.txt” powinien zawierać polskie znaki wprowadzone w kodzie Cp1250: "ąśęółżźćń"; */

public class Czytaj{

    public static void main(String args[]){
      String s;

      char zn[] = new char[9];
      try{
     InputStreamReader r = new InputStreamReader((new FileInputStream("plik.txt")), "Cp1250");
     r.read(zn,0,zn.length);
    s = new String(zn);
    System.out.println(s);
            OutputStreamWriter o = new OutputStreamWriter((new FileOutputStream("plik1.txt")),"Cp852");
    o.write(s,0,s.length());
    o.flush();
    } catch (Exception e){}

    }

}// koniec public class Czytaj
```

Prezentowany program tworzy dwa strumienie: jeden wejścia czytający plik tekstowy zapisany według strony tekstowej Cp1250 (Windows PL) i drugi wyjścia zapisujący nowy plik wyjściowy tekstem według strony kodowej Cp852 (DOS PL). Oczywiście nazwy plików jak i nazwy stron kodowych można zmieniać dla potrzeb ćwiczeń i w ten sposób dokonywać konwersji plików z np. Cp1250 na ISO8859-2. W celu weryfikacji działania powyższego programu należy otworzyć plik o nazwie „plik.txt” w edytorze obsługującym Cp1250 a plik o nazwie „plik1.txt” w edytorze obsługującym Cp852. W przypadku gdy nie ma konieczności zmiany systemu kodowania znaków warto wykorzystywać klasy *FileReader* oraz *FileWriter* zamiast *InputStreamReader* i *OutputStreamWriter*. Dla poprawienia efektywności pracy istotne jest również buforowanie czytanych danych co w rezultacie prowadzi do następującego wywołania obiektu:

```
BufferedReader br = new BufferedReader(new FileReader(”plik.txt”));
```

7.4 Dzielenie strumienia – klasa StreamTokenizer

Na zakończenie omawiania klas związanych z obsługą strumieni warto zapoznać się z klasą *StreamTokenizer*, dzieląca strumień tekstowy na leksemy. Klasa ta daje więc swoistą funkcjonalność wykrywanie elementów strumienia i umieszczania ich w tablicy. Wskazując znak oddzielający leksemy można dokonać przeformatowania przesłanego tekstu (np. podzielić ścieżkę dostępu, dokonać detekcji liczb w tekście, itp.). Pobranie leksemu z tablicy odbywa się poprzez wywołanie metody *nextToken*(). Poniższy przykład ilustruje stosowanie klasy *StreamTokenizer*:

Przykład 7.5:

```
//FormatujStrumien.java:

import java.io.*;

public class FormatujStrumien{

	public static void main(String args[]){
		System.out.println("Podaj tekst zawierający znak : .");
		 Reader r = new BufferedReader(new InputStreamReader(System.in));
		StreamTokenizer st = new StreamTokenizer(r);
		st.ordinaryChar('.');
		st.ordinaryChar('-');
		st.ordinaryChar('A');
		try{
			while (st.nextToken() != StreamTokenizer.TT_EOF){
				if (st.ttype==StreamTokenizer.TT_WORD){
					System.out.println(new String(st.sval));
				}

			}
		}catch (IOException ioe){
			System.out.println("Błąd operacji I/O");
		}

	}


}//koniec public class FormatujStrumien
```

W powyższym przykładzie stworzono obiekt klasy *Reader* na podstawie standardowego strumienia wejścia, który jest następnie wykorzystany przy inicjowaniu obiektu klasy *StreamTokenizer*. Ponieważ w zbiorze domyślnych znaków dzielących strumień nie ma znaków '.' oraz '-' dodano te znaki za pomocą metody *ordinaryChar*(). Dodatkowo ustawiono znak 'A' jako znak dzielący strumień. Następnie w bloku instrukcji warunkowej uruchomiona jest pętla działająca tak długo aż nie wystąpi koniec pliku (aż nie wciśnięty zostanie kod CTRL-Z a następnie

Enter). W pętli pobierany jest kolejny wczytywany element, sprawdzany jest jego typ i jeśli jest to słowo (tekst) to drukowany jest na ekranie tekst będący wartością pola *sval* obiektu klasy *StreamTokenizer*. Obsługa programu polega na wprowadzaniu tekstu ze znakami go dzielącymi i kończeniu linii wciskając *Enter*. W ten sposób linia po linii można analizować wprowadzany tekst. Koniec pracy programu jest wymuszany sekwencją końca pliku CTRL-Z i *Enter*.

Warto zauważyć, że istnieje klasa *StringTokenizer* o podobnym działaniu, której argumentem nie jest jednak strumień a obiekt klasy *String*.

7.5 Strumienie poza java.io

W JDK istnieją jeszcze inne strumienie zdefiniowane poza pakietem *java.io*. Przykładowo w pakiecie *java.util.zip* zdefiniowano szereg klas strumieni obsługujących kompresję w formie ZIP i GZIP. Podstawowe klasy strumieni tam przechowywane to:

CheckedInputStream
CheckedOutputStream
DeflaterOutputStream
GZIPInputStream
GZIPOutputStream
InflaterInputStream
ZipInputStream
ZipOutputStream

Przykładowo w celu dokonania kompresji pliku metodą GZIP można zastosować następujący kod:

Przykład 7.6:

```
//Kompresja.java:

import java.io.*;
import java.util.zip.*;

public class Kompresja{

        public static void main(String args[]){
          String s;

        byte b[] = new byte[100];
        for (int i=0; i<100; i++){
                b[i]=(byte) (i/10);
        }

         try{
                FileOutputStream o = new FileOutputStream("plik2.txt");
                o.write(b);
                o.flush();
                o.close();
```

```
            FileOutputStream fos = new FileOutputStream("plik2.gz");
            GZIPOutputStream z = new GZIPOutputStream(new BufferedOutputStream(fos));
            z.write(b,0,b.length);
            z.close();

        } catch (Exception e){}

  }

}// koniec public class Kompresja
```

W prezentowanym kodzie tworzona jest tablica bajtów wypełniana kolejnymi wartościami od 1 do 10. Tablica ta jest następnie przesyłana do strumieni wyjściowych raz bezpośrednio do pliku, drugi raz poprzez kompresję metodą GZIP. W wyniku działania programu uzyskuje się dwa pliki: bez kompresji „plik2.txt" i z kompresją „plik2.gz".

W pakietach standardowych jak i w pakietach będących rozszerzeniem bibliotek Javy można znaleźć jeszcze szereg innych strumieni związanych z przesyłaniem danych (np. w kryptografii czy w obsłudze portów).

Ze względu na liczne potrzeby wykorzystywania portów szeregowych i równoległych komputera Sun opracował pakiet rozszerzenia Javy o nazwie javax.comm. Pakiet ten umożliwia obsługę portów poprzez strumienie. Poniższy przykład ukazuje próbę zapisu do portu szeregowego.

Przykład 7.7:

```
//ZapiszPort.java:

import java.io.*;
import java.util.*;
import javax.comm.*;

public class ZapiszPort {
  static SerialPort port;
  static CommPortIdentifier id;
  static Enumeration info;
  static String dane = "Tu Czerwona Jarzębina - odbiór \n";
  static OutputStream os;

   public static void main(String args[]) {
      info = CommPortIdentifier.getPortIdentifiers();

      while (info.hasMoreElements()) {
         id = (CommPortIdentifier) info.nextElement();
         if (id.getPortType() == CommPortIdentifier.PORT_SERIAL) {
            if (id.getName().equals("COM1")) {
                try {
                   port = (SerialPort) id.open("ZapiszPort", 2000);
               } catch (PortInUseException e) {}
               try {
```

```
                os = port.getOutputStream();
            } catch (IOException ioe) {}
            try {
                  port.setSerialPortParams(9600,
                  SerialPort.DATABITS_8,
                  SerialPort.STOPBITS_1,
                  SerialPort.PARITY_NONE);
            } catch (UnsupportedCommOperationException ue) {}
            try {
                os.write(dane.getBytes());
            } catch (IOException e) {}
          }
        }
      }
    }
}// koniec public class ZapiszPort
```

Rozdział 8 Integracja Javy z innymi językami - JNI. Programowanie sieciowe

8.1 Integracja Javy z innymi językami - Java Native Interface (JNI)
8.1.1 Obsługa metod rodzimych w kodzie Javy
8.1.2 Kompilacja i generacja plików nagłówkowych
8.1.3 Implementacja metody rodzimej - funkcja a biblioteka
8.1.4 Dostęp do metod i pól zdefiniowanych w Javie
8.2 Programowanie sieciowe
8.2.1 Adresowanie komputerów w sieci (InetAddress i URL)
8.2.2 Komunikacja przez Internet (klient-serwer)
8.2.3 Inne mechanizmy programowania sieciowego w Javie

8.1 Integracja Javy z innymi językami - Java Native Interface (JNI)

Tworząc programy w środowisku języka programowania Java napotyka się czasami na ograniczenia związane z dostępem do specyficznych dla danej platformy cech. Konieczne jest wówczas zastosowanie narzędzi obsługujących te cechy a następnie zaimplementowanie tych narządzi w kodzie programu tworzonego w Javie. Operacja taka jest możliwa poprzez wykorzystanie swoistego interfejsu pomiędzy kodem w Javie a kodem programu stworzonego w innym środowisku, np. C lub C++. Co więcej wykorzystanie istniejących już funkcji (napisanych wcześniej w innych językach programowania niż Java) może znacznie uprościć tworzenie nowej aplikacji w Javie, szczególnie wtedy gdy liczy się czas. Interfejs umożliwiający to specyficzne połączenie kodów został nazwany Java Native Interface, lub w skrócie JNI. Każda funkcja napisana w innym języku niż Java a implementowana bezpośrednio w kodzie Javy nosi nazwę metody rodzimej („native method”) i wymaga jawnej, sformalizowanej deklaracji. Metody rodzime mogą wykorzystywać obiekty Javy tak, jak to czynią metody tworzone w Javie, a więc tworzyć obiekty, używać je oraz modyfikować. Aby funkcjonalność metod rodzimych była pełna metody te mogą wywoływać metody tworzone w Javie, przekazywać im parametry i pobierać wartości lub referencje zwracane przez te metody. Możliwa jest również obsługa wyjątków metod rodzimych.
Czym jest więc JNI? JNI to interfejs zawierający:

- plik nagłówkowy środowiska rodzimego (np. plik jni.h dla środowiska C);
- generator pliku nagłówkowego metody rodzimej (np. javah -jni);
- formalizację deklaracji metody rodzimej,
- definicję i rzutowanie typów danych,
- zbiór metod (funkcji) umożliwiających wymianę danych i ustawianie stanów (np. wyjątków, monitora, itp.).

Implementacja JNI polega najprościej na wykonaniu następujących działań:

1. stworzenie programu w Javie zawierającego deklarację metody rodzimej (native);
2. kompilacja programu;
3. generacja pliku nagłówkowego środowiska rodzimego dla klasy stworzonego programu (javah -jni);
4. stworzenie implementacji metody rodzimej z wykorzystaniem plików nagłówkowych interfejsu JNI i klasy stworzonego programu ;
5. kompilacja metody rodzimej i umieszczenie jej w bibliotece;

6. uruchomienie programu korzystającego z metody rodzimej poprzez ładowanie biblioteki.

8.1.1 Obsługa metod rodzimych w kodzie Javy

Pierwszym krokiem w implementacji interfejsu JNI jest stworzenie kodu w Javie obsługującego metody rodzime. Najprostsza struktura obsługi może wyglądać następująco:

Przykład 8.1:

```
//Informacje.java:

class Informacje{
	//deklaracja metody rodzimej
	public native int infoSystemu(String parametr);

	//ładowanie biblioteki zawierającej implementację metody rodzimej
	static{
		System.loadLibrary(”sysinfo”);
	}

	//wykorzystanie metody rodzimej
	public static void main(String args[]){
		Informacje i = new Informacje();
		int status = i.infoSystemu(”CZAS”);
	}
}// koniec class Informacje
```

Powyższy szkic stosowania metod rodzimych zawiera trzy bloki. Pierwszy z nich deklaruje metodę rodzimą, która różni się od pozostałych metod tym, że używany jest specyfikator „native” w deklaracji. Drugi blok to kod statyczny ładowany przy interpretowaniu (kompilacji) kodu bajtów pobierający bibliotekę przechowujący realizację metody rodzimej. Ostatni blok to zastosowanie metody rodzimej. Zadeklarowanie metody jako „native” oznacza, że kompilator ma uznać daną metodę jako rodzimą zdefiniowaną i zaimplementowaną poza Javą. Podana w deklaracji metody rodzimej nazwa jest odwzorowywana później na nazwę funkcji w kodzie rodzimym zgodnie z regułą:
nazwa -> Java_NazwaPakietu_NazwaKlasy_nazwa, czyli np.
infoSystemu -> Java_Informacje_infoSystemu, (brak nazwy pakietu, gdyż klasa Informacje zawiera się w pakiecie domyślnym, który nie posiada nazwy).
Wykorzystanie bibliotek, w których znajduje się realizacja metod rodzimych wymaga, aby biblioteki te były dostępne dla uruchamianego programu, tzn. musi być odpowiednio ustalona ścieżka dostępu.

8.1.2 Kompilacja i generacja plików nagłówkowych

Kompilacja kodu Javy wykorzystującego metody rodzime odbywa się tak samo jak dla czystego kodu Javy, np. javac -g Informacje.java.

Nowością jest natomiast wygenerowanie pliku nagłówkowego, jaki zawarty będzie w kodzie metody rodzimej. Generacja taka wymaga zastosowania narzędzia javah, które generuje plik nagłówkowy o nazwie takiej jak podana nazwa klasy z rozszerzeniem .h (header - nagłówek) tworzony w tym samym katalogu gdzie znajduje się plik klasy programu. Przykładowo wywołanie polecenia:

```
javah -jni Informacje
```

spowoduje wygenerowanie następującego pliku nagłówkowego:

Przykład 8.2:

```
//Informacje.h:

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Informacje */

#ifndef _Included_Informacje
#define _Included_Informacje
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Informacje
 * Method:    infoSystemu
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_Informacje_infoSystemu
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Jak widać jest to nagłówek dla kodu metody rodzimej tworzonego w C/C++. Zasadniczo można tworzyć implementacje metod rodzimych w innych językach ale to wymaga indywidualnego podejścia (konwersji typów i nazw), stąd zaleca się korzystanie jedynie z C/C++ do obsługi metod rodzimych. W zasadniczej części nagłówka (poza analizą składni dla C czy C++) zawarto opis metody używając do tego komentarza zawierającego nazwę klasy, w ciele której zadeklarowano metodę rodzimą, nazwę metody rodzimej w kodzie Javy oraz podpis (sygnatura) metody. Sygnatura metody ma format *(typy-argumentów)typy-zwracane*; gdzie poszczególne typy są reprezentowane przez swoje sygnatury, i tak:

sygnatura	znaczenie typu w Javie
Z	**boolean**
B	**byte**
C	**char**
S	**short**
I	**int**

J	**long**
F	**float**
D	**double**
Lpełna-nazwa-klasy	**pełna nazwa klasy**
[typ	**typ[]**

W powyższym przykładzie pliku nagłówkowego widnieje informacja, że metoda zawiera argument o sygnaturze *Ljava/lang/String* czyli obiekt klasy *String* oraz zwraca wartość typu o sygnaturze *I* czyli typu *int*. Można określić sygnatury typów argumentów i wartości zwracanych metod poprzez użycie narzędzia de-asemblacji kodu a mianowicie *javap*:

javap -s -p Informacje

co wygeneruje:

```
Compiled from Informacje.java
class Informacje extends java.lang.Object {
   static {};
      /*   ()V   */
   Informacje();
      /*   ()V   */
   public native int infoSystemu(java.lang.String);
      /*   (Ljava/lang/String;)I   */
   public static void main(java.lang.String[]);
      /*   ([Ljava/lang/String;)V   */
}
```

W powyższym wydruku w opcji komentarza zawarto sygnatury typów.

Po opisie metody rodzimej następuje deklaracja metody dla środowiska rodzimego czyli C/C++. Deklaracja ta oprócz omówionej już nazwy zawiera szereg nowych elementów i tak:

JNIEXPORT i JNICALL - oznaczenie funkcji eksportowanych z bibliotek (jest to odniesienie się do specyfikacji deklaracji funkcji eksportowanych, JNIEXPORT oraz JNICALL są zdefiniowane w jni_md.h);

jint lub inny typ - oznaczenie typu elementu zwracanego przez funkcję, np.

typ w Javie,	**typ rodzimy,**	**rozmiar,**	**typ dla C/C++ (Win32)**
boolean	**jboolean**	**8**	**unsigned unsigned char**
byte	**jbyte**	**8**	**signed char**
char	**jchar**	**16**	**unsigned unsigned short**
short	**jshort**	**16**	**short**
int	**jint**	**32**	**long**
long	**jlong**	**64**	**__int64**
float	**jfloat**	**32**	**float**
double	**jdouble**	**64**	**double**
void	**void**		**void**

*JNIEnv** wskaźnik interfejsu JNI zorganizowany jako tablica funkcji JNI o konkretnej lokalizacji. Metoda rodzima wykorzystuje funkcje poprzez odwołanie się do wskaźnika JNIEnv*. Przykładowo można pobrać rozmiar macierzy wykorzystując funkcję GetArrayLength() poprzez wskaźnik JNIEnv*:

```
(...) JNIEnv *env (...) jintArray macierz (...)
jsize rozmiar = (*env)->GetArrayLength(env, macierz);
```

jobject to kolejny element deklaracji funkcji w JNI. Argument tego typu stanowi referencję do bieżącego obiektu; jest odpowiednikiem this w Javie.
Warto tu zauważyć, że odpowiednikiem metody Javy zadeklarowanej jako ”native” jest funkcja zawierająca co najmniej dwa argumenty, nawet wówczas gdy metoda nie zawiera żadnego.

jstring lub inne typy argumentów - kolejne argumenty funkcji (argumenty metody rodzimej).

8.1.3 Implementacja metody rodzimej - funkcja a biblioteka

Kolejny krok stosowania funkcji w kodzie Javy to stworzenie tych funkcji lub ich adaptacja do formy wymaganej przez JNI. Deklaracja realizowanej funkcji musi być taka sama jak założona (wygenerowana) w pliku nagłówkowym. Następujący przykład ukazuję przykładową realizację metody rodzimej deklarowanej we wcześniejszych przykładach:

Przykład 8.3:

```
//informacje.cpp

#include "jni.h"
 #include <stdio.h>
 #include <dos.h>
 #include "Informacje.h"


JNIEXPORT jint JNICALL Java_Informacje_infoSystemu(JNIEnv *env, jobject o, jstring str){
   struct  time t;
   const char *s=(env)->GetStringUTFChars(str,0);
   printf("Obsługiwana aktualnie opcja to: %s\n",s);
   if ((((*s=='C')&&(*(s+1)=='Z'))&&(*(s+2)=='A'))&&(*(s+3)=='S')){
      gettime(&t);
           printf("Bieżący czas to: %2d:%02d:%02d.%02d\n",t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
      return 1;
   } else {
           printf("Nie obsługiwana opcja");
      return 0;
   }
}
```

W powyższym kodzie funkcji w C++ włączono szereg plików nagłówkowych. Plik *stdio.h* oraz *dos.h* to standardowe pliki nagłówkowe biblioteki C. Pierwszy jest wymagany ze względu na stosowanie funkcji *printf*, drugi ze względu na dostęp do czasu systemowego poprzez funkcję *gettime()* oraz strukturę *time*. Ponadto zawarto dwa pliki nagłówkowe wymagane ze względu na implementację interfejsu JNI, czyli jni.h (definicja typów i metod) oraz Informacje.h (deklaracja funkcji odpowiadającej metodzie rodzimej). Pierwsze linia kodu funkcji zawiera deklarację zmiennej typu struktury *time* przechowującej informacje związane z czasem systemowym po wykorzystaniu funkcji *gettime*() w dalszej części kodu. Kolejna linia kodu jest bardzo ważna ze względu na zobrazowanie działania konwersji danych. Wykorzystano tam funkcję *GetStringUTFChars*() do konwersji zmiennej (argumentu funkcji) typu *jstring* do *const char **. Konwersja ta jest wymagana gdyż nie występuje wprost odwzorowanie typów *String* na *const char *.* Jest to również spowodowane tym, że Java przechowuje znaki w Unicodzie i dlatego konieczna jest dodatkowa konwersja znaków z typu *String*(Unicode -UTF) na *char* *. Warto zwrócić uwagę na wywołanie funkcji konwertującej. Odbywa się ono poprzez wykorzystanie wskaźnika (*env*) do struktury przechowującej szereg różnych funkcji (określonych w jni.h). Ze względu na sposób zapisu funkcji w jni.h możliwe są dwa sposoby wywołania funkcji albo:
(env*)->GetStringUTFChars(env*, str ,0) dla C albo:
(env)->GetStringUTFChars(str,0) dla C++.
Obie funkcje są zapisane w jni.h, druga z nich jest realizowana poprzez zastosowanie pierwszej z pierwszym argumentem typu *this*.
Dalsza część kodu przykładu to wyświetlenie komunikatu zawierającego wartość argumentu tekstowego po konwersji, pobranie informacji o czasie systemowym i wyświetlenie go.
Tak przygotowany kod funkcji należy następnie skompilować i wygenerować bibliotekę (w prezentowanym przykładzie będzie to biblioteka typu Dynamic Link Library - DLL).
Należy pamiętać o lokalizacji plików jni.h oraz innych plików nagłówkowych wołanych przez jni.h. Pliki te są zapisane w podkatalogu *include* oraz *include/win32* głównego katalogu JDK.

Tak przygotowana funkcja i biblioteka umożliwiają uruchomienie programu w Javie wykorzystującego metodę rodzimą. W wyniku wywołania prezentowanego na początku kodu :
java Informacje
uzyskamy rezultat typu:

Obsługiwana aktualnie opcja to: CZAS
Bieżący czas to: 16:11:06.50

gdzie podawany czas zależy oczywiście od ustawień systemowych.

8.1.4 Dostęp do metod i pól zdefiniowanych w Javie

W celu wykorzystania w funkcji metod i metod statycznych zdefiniowanych w Javie należy wykonać trzy podstawowe kroki:
1. pobrać obiekt klasy (Class) w ciele której znajdować ma się żądana metoda:
jclass c = (env)->GetObjectClass(o); gdzie o jest zmienną typu jobject

2. pobrać identyfikator metody dla danej klasy poprzez podanie nazwy metody oraz sygnatur:
jmethodID id = (env)->GetMethodID(c, ”suma”, ”(II)I”); id przyjmuje wartość 0 jeżeli nie ma szukanej metody w klasie reprezentowanej przez obiekt c,
3. wywołać metodę poprzez podanie identyfikatora id oraz obiektu o, np.:
(env)->CallIntMethod(o,id,a1,a2); gdzie a1 i a2 to argumenty. W zależności od zwracanego typu można wykorzystać różne funkcje np. CallVoidMethod(), CallBooleanMethod(), itp.

Wykorzystanie metod statycznych jest analogiczne do powyższego schematu z tą różnicą, że w kroku 2 i 3 należy wywołać odpowiednie funkcje *GetStaticMethodID*() i *CallStaticIntMethod*() (lub podobne innych typów), przy czym funkcje CallStatic* jako argument wykorzystują zmienną typu *jclass* zamiast *jobject* (odwołanie się do obiektu klasy *Class* reprezentującego klasę zamiast do obiektu będącego wystąpieniem klasy).

Przykładowe fragmenty kodów w Javie i w C obrazujące stosowanie metod mogą być następujące:

```
//Liczenie.java:

(...)
public native void oblicz(int a, int b);
public int suma(int a, int b){
        return (a+b);
}
(...)

//policz.cpp:

(...) JNIEXPORT void JNICALL Java_Liczenie_oblicz(JNIEnv *env, jobject o, jint a, jint b){
        jclass c = (env)->GetObjectClass(o);
        jmethodID id =  (env)->GetMethodID(c, ”suma”, ”(II)I”);
        if (id==0)
                return;
        printf(„Oto wartość sumy argumentów: %d”, (env)->CallIntMethod(o,id,a,b));
}
```

Dostęp do pól obiektów i zmiennych statycznych klas Javy z poziomu funkcji jest wykonywany w podobny sposób jak dla metod. Wyróżnia się trzy kroki postępowania:

1. pobrać obiekt klasy (Class) w ciele której znajdować ma się żądana zmienna:
jclass c = (env)->GetObjectClass(o); gdzie o jest zmienną typu jobject
2. pobrać identyfikator zmiennej dla danej klasy poprzez podanie nazwy zmiennej oraz sygnatury:
jfielddID id = (env)->GetFieldID(c, ”a”, ”I”); id przyjmuje wartość 0 jeżeli nie ma szukanej zmiennej w klasie reprezentowanej przez obiekt c,
3. pobrać lub ustawić wartość zmiennej poprzez podanie identyfikatora id oraz obiektu o, np.:
jint a = (env)->GetIntField(o,id);
lub
(env)->SetIntField(o,id,a);

W zależności od typu zmiennej można wykorzystać różne funkcje np. SetObjectField(), GetObjectField(), GetLongField(), SetDoubleField(), itp.

Wykorzystanie pól statycznych jest analogiczne do powyższego schematu z tą różnicą, że w kroku 2 i 3 należy wywołać odpowiednie funkcje *GetStaticFieldID*() i *SetStaticIntField*() (lub podobne innych typów), przy czym funkcje *{Set,Get}Static** jako argument wykorzystują zmienną typu jclass zamiast jobject (odwołanie się do obiektu klasy *Class* reprezentującego klasę zamiast do obiektu będącego wystąpieniem klasy).

Przykładowe fragmenty kodów w Javie i w C ukazujące dostęp do pól mogą być następujące:

```
//Liczenie.java:
class Liczenie{
static String str=”Operacja zakonczona”;
int suma = 10;
(...)
public native void oblicz(int a, int b);
public int suma(int a, int b){
        return (a+b);
}
(...)
}

//policz.cpp:

(...) JNIEXPORT void JNICALL Java_Liczenie_oblicz(JNIEnv *env, jobject o, jint a, jint b){
        jclass c = (env)->GetObjectClass(o);
        jfieldID id =  (env)->GetFieldID(c, ”suma”, ”I”);
        if (id==0)
                return;
        jint s = (env)->GetIntField(o,id);
        printf(„Oto wartość sumy argumentów: %d”,s);
        (env)->SetIntField(o,id, (env)->CallIntMethod(o,id,a,b));
        id =  (env)->GetStaticFieldID(c, ”str”, ”Ljava/lang/String;”);
        if (id==0)
                return;
        jstring tekst = (env)->GetStaticObjectField(c,id);
        printf(tekst);

}
```

Na zakończenie tej sekcji warto przedstawić w skrócie zagadnienia związane z wyjątkami. Wyjątki mogą pojawić się w różnych okolicznościach w pracy z metodami rodzimymi np. przy braku wzywanej metody, przy braku wzywanego pola, przy złym argumencie, itp. JNI dostarcza klika funkcji do obsługi zdarzeń w kodzie funkcji. Podstawowe funkcje to *ExceptionOccurred*(), *ExceptionDescribe*() i *ExceptionClear*(). Pierwsza funkcja bada czy wystąpił wyjątek; jeśli tak to zwraca wartość logiczną odpowiadającą prawdzie. Wówczas można w bloku instrukcji warunkowej zawrzeć pozostałe dwie funkcje (wysłanie informacji o wyjątku na ekran, oraz wyczyszczenie wyjątku). Dodatkowo można wywołać wyjątek, który zostanie zwrócony do kody Javy,

gdzie należy go obsłużyć. Można to zrobić za pomocą *funkcji Throw(jthrowable o)* lub *ThrowNew(jclass, const char *)*.

Należy pamiętać również o tym, że JNI umożliwia korzystanie z Maszyny Wirtualnej w ramach aplikacji rodzimej. W tworzeniu takiej aplikacji zanim wywoływane będą metody i pola (tak jak to ukazano do tej pory) konieczne jest wywołanie Maszyny Wirtualnej Javy oraz uzyskanie obiektu klasy, np.:

```
JDK1_1InitArgs vm;
JavaVM *jvm;
JNIEnv *env;
jclass c;

vm.version=0x00010001;
//określenie wersji Maszyny Wirtualnej jako 1.1.2 i wyższych
JNI_GetDefaultJavaVMInitArgs(&vm);
jint test = JNI_CreateJavaVM(&jvm,&env,&vm);

c=(env)->FindClass("NazwaKlasy");

// np. GetStatic{Method,Field}Int(c,...) i tak dalej
```

8.2 Programowanie sieciowe

Tworzenie programów działających w sieciach komputerowych było jednym z celów stworzenia języka Java. Możliwości aplikacji sieciowych tworzonych za pomocą Javy są różne. Podstawowe, proste rozwiązania wykorzystujące mechanizmy wysokiego poziomu pracy w sieci dostarczane są przez podstawowe klasy połączeń (adresowania) jak *InetAddress* i *URL*.

8.2.1 Adresowanie komputerów w sieci (InetAddress i URL)

Klasa *InetAddress* opisuje adres komputera w sieci poprzez nazwę/domenę, np. *www-med.eti.pg.gda.pl* oraz poprzez numer IP, np. 153.19.51.66. Istnieje szereg metod statycznych klasy *InetAddress* wykorzystywanych do tworzenia obiektu klasy, brak jest bowiem konstruktorów. Podstawowe metody wykorzystywane do tworzenia obiektów to:

```
InetAddress.getByName(String nazwa),
InetAddress.getAllByName(String nazwa),
InetAddress.getLocalHost();
```

Pierwsza metoda tworzy obiekt klasy bazując na podanej nazwie komputera lub adresie. Druga metoda jest wykorzystywana wówczas kiedy komputer o danej nazwie ma wiele adresów IP. Zwracana jest wówczas tablica obiektów typu *InetAddress*. Ostania metoda jest wykorzystywana do uzyskania obiektu reprezentującego adres komputera lokalnego. Wszystkie metody muszą zawierać deklaracje lub obsługę wyjątku *UnknownHostException* powstającego w przypadku

braku identyfikacji komputera o podanej nazwie lub adresie. Poniżej zaprezentowano przykładowy program wykorzystujący prezentowane metody klasy *InetAddress*.

Przykład 8.4:

```
//Adresy.java

//Adresy.java

import java.net.*;

public class Adresy{

 public static void main(String args[]){

   try{
     InetAddress a0 = InetAddress.getLocalHost();
     System.out.println("Adres komputera "+a0.getHostName()+" to: " +a0);
     InetAddress a1 = InetAddress.getByName("biomed.eti.pg.gda.pl");
     System.out.println("Adres komputera biomed to: "+a1);
     InetAddress a2[] = InetAddress.getAllByName("www.eti.pg.gda.pl");
     System.out.println("Adres komputera www.eti.pg.gda.pl to:");
     for(int i=0; i<a2.length; i++){
            System.out.println(a2[i]);
     }
   } catch (UnknownHostException he) {
     he.printStackTrace();
   }

 }

}// koniec public class Adresy
```

W wyniku działania powyższego programu wyświetlone zostaną informacje na temat adresów sieciowych wybranych komputerów. W programie zastosowano również jedną z metod klasy *InetAddress* umożliwiającą operacje na adresie a mianowicie *getHostName*(). Metoda ta zwraca nazwę komputera jako obiekt klasy String. Istnieją również metody umożliwiające filtrację adresu komputera w celu uzyskania jedynie numeru IP: *byte[] getAddress(), String getHostAddress().*

Inną klasą wykorzystywaną w Javie do adresowania komputerów jest klasa *URL* oraz jej pochodne (*URL, URLClassLoader, URLConnection, URLDecoder, URLEncoder, URLStreamHandler). URL* czyli Uniform Resource Locator jest specjalną formą adresu zasobów w sieci. *URL* posiada dwa podstawowe elementy: identyfikator protokołu oraz nazwę zasobów. Identyfikator protokołu to np*. http, ftp, gopher, rmi* czy *jdbc*. Nazwę zasobów stanowią takie elementy jak: nazwa hosta, nazwa pliku, numer portu, nazwa odwołania (w danym pliku). Tworząc więc obiekt klasy URL otrzymujemy gotowy wskaźnik, który jest wykorzystywany przez liczne metody Javy (np. otwieranie obrazka *getImage*(), tworzenie połączenia w JDBC - Connection). W odróżnieniu od klasy *InetAddress* tworzenie obiektów klasy *URL* odbywa się poprzez wykorzystanie jednego z licznych konstruktorów. Każdy z nich związany jest z koniecznością obsługi wyjątku *MalformedURLException* powstającym w wypadku problemów z identyfikacją wskazanego w wywołaniu konstruktora

protokołu. Liczne konstruktory umożliwiają podania parametrów adres URL albo poprzez odpowiednik adresu będący tekstem (*String*) albo poprzez tworzenie tego adresu z takich elementów jak nazwa protokołu, nazwa komputera, port, nazwa pliku, itp. Przykładowe konstruktory mają postać:

URL(String adres) ,
URL(String protokół, String host, int port, String plik)

Klasa *URL* zawiera szereg metod umożliwiających filtrację adresu, a więc pobranie nazwy protokołu (*getProtocol*()), nazwy komputera (*getHost*()), pliku (*getFile*()) czy numeru portu (*getPort*()). Dodatkowo klasa *URL* zawiera metody umożliwiające wykonywanie połączenia z hostem (tworzone jest gniazdo – o czym dalej) i przesyłanie danych. Przykładowy program ukazuje możliwość wykorzystania funkcjonalności klasy *URL*:

Przykład 8.5:

```
//Pobiez.java

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

public class Pobiez{

 public static void main(String args[]){
   URL url;
   String tekst;
   if (args.length !=1) {
           System.out.println("Wywołanie: Pobiez URL; gdzie URL to adres zasobów");
           System.exit(1);
   }
   try{
           url = new URL(args[0]);
           BufferedReader br = new BufferedReader(new InputStreamReader(url.openStream()));
           while( (tekst=br.readLine()) !=null){
                   System.out.println(tekst);
           }
   } catch (Exception e) {
           e.printStackTrace();
   }

 }

}// koniec public class Pobiez
```

Powyższy program umożliwia pobranie źródła wskazanego pliku (*html*) i wyświetlenie go na ekranie. Można oczywiście tak skonstruować strumień aby pobierany plik był nagrywany do lokalnego pliku i w ten sposób stworzyć narzędzie do kopiowania stron z Internetu. W przykładzie zastosowano konstrukcję obiektu klasy *URL* poprzez podanie argumentu do konstruktora jako tekstu (*String*) będącego argumentem wywołania aplikacji. Przykładowe wywołanie aplikacji może być następujące:

java Pobiez http :// biont.eti.pg.gda.pl/java.htm

w wyniku czego uzyskamy wydruk:

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
  <meta name="Author" content="Jacek Ruminski">
  <meta name="GENERATOR" content="Mozilla/4.6 [en-gb] (WinNT; I) [Netscape]">
  <title>Jezyk JAVA - wyklad</title>
</head>
<body text="#FFFFFF" bgcolor="#330033" link="#0000EE" vlink="#551A8B" alink="#FF0000">

<center><b><i><font size=+3>Jŕzyk <img SRC="javalogo.gif" height=88 width=52 align=ABSCENTER>
wyk│ad</font></i></b>
<br> 
<p><img SRC="jav_logo.jpg" height=125 width=260>
<p><b><font color="#CC0000"><font size=+1>Plan wyk│adu:</font></font></b></center>

(...)
```

8.2.2 Komunikacja przez Internet (klient-serwer)

Do komunikacji poprzez Internet programy Javy wykorzystują protokoły TCP i UDP. Klasy *URL** oraz *Socket* i *ServerSocket* wykorzystują Transfer Control Protocol klasy takie jak *DatagramPacket*, *DatagramSocket*, oraz *MulticastSocket* korzystają z User Datagram Protocol. W pakiecie java.net zdefiniowane są jeszcze inne klasy, z których warto przytoczyć te związane z autoryzacją połączeń i nadawaniem uprawnień: *Authenticator*, *NetPermission*, *PasswordAuthentication*, *SocketPermission*.

W aplikacjach klient-serwer, serwer dostarcza określonej usługi np. przetwarza zapytanie skierowane do bazy danych, zapisuje serię obrazów diagnostycznych. Klient wykorzystuje usługi świadczone przez serwer i jest odpowiedzialny za żądanie usługi oraz obsługę wyników. Funkcje pełnione przez każdą ze stron można ująć następująco:
- połączenie z urządzeniem zdalnym (przygotowanie wysyłania i odbioru danych),
- wysyłanie danych,
- odbiór danych,
- zamknięcie połączenia,
- przywiązanie portu (dla danej aplikacji na danym hoście),
- nasłuch,
- akceptacja połączeń na danych portach.

Pierwsze cztery funkcje są właściwe dla klienta, serwer rozszerza je o dodatkowe trzy. W Javie obsługę klienta, a więc realizację pierwszych czterech funkcji dostarcza klasa *Socket*; dla serwera przeznaczono klasę *ServerSocket*. *Socket* (gniazdo) to dobrze znane pojęcie wynikające z planów stworzenia dostępu do sieci jako do strumienia danych (strumień wejścia, strumień wyjścia). Koncepcja strumieni miała być uniwersalna dla każdego możliwego przepływu danych (z urządzenia, z pliku, z

sieci). *Socket* jest więc pewną abstrakcją umożliwiającą przejście na wyższy poziom i nie zajmowaniem się takimi zagadnieniami jak rozmiar pakietu, retransmisja pakietu, typ mediów, itp. Rozważmy teraz zagadnienia związane ze stroną klienta procesu komunikacji sieciowej, a więc zapoznajmy się z klasą *Socket* pakietu java.net.

Klasa *Socket* posiada szereg różnych konstruktorów, z których najbardziej popularne są:
Socket(InetAddress address, int port) - gdzie address to obiekt klasy InetAddress będący adresem IP lub nazwą hosta, port - numer portu (0-65535),
Socket(String host, int port) - gdzie host to tekst oznaczający nazwę hosta, port - numer portu (0-65535).

Ponieważ połączenie wykonywane przez obiekt klasy *Socket* może nie powieść się z różnych przyczyn (np. nie znany host) konieczna jest obsługa wyjątków.
Klasyczny fragment kodu obsługi klasy *Socket* pokazano poniżej:

```
try {
      Socket  gniazdo= new Socket("www.amg.gda.pl", 80);
    }
    catch (UnknownHostException e) {
      System.err.println(e);
    }
    catch (IOException e) {
      System.err.println(e);
   }
```

W powyższym przykładzie podjęta jest próba (*try*) połączenia z serwerem www.amg.gda.pl na porcie 80. Jeżeli nazwa hosta jest nieznana lub serwer nazw nie działa zostanie zwrócony wyjątek *UnknownHostException* (wyjątek nieznanego hosta). Jeśli z innych przyczyn nie uda się uzyskać połączenia zostanie zwrócony wyjątek *IOException* (wyjątek wejścia-wyjścia). Najprostszym przykładem programu implementującego klasę *Socket* może być prosty skaner portów serwera:

Przykład 8.6:

```
//SkanerPortow.java:

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


public class SkanerPortow {

  public static void main(String[] args) {

    Socket gniazdo;
    String host = "localhost";

    if (args.length > 0) {
      host = args[0];    //jeśli nie podano argumentu programu hostem bedzie komputer lokalny
    }
    for (int n = 0; n < 1024; n++) {   //skanuj wszystkie porty "roota"
```

```
    try {
      gniazdo = new Socket(host, n);
      System.out.println("Znalazłem serwer na porcie " + n + " komputera: " + host);
    }
    catch (UnknownHostException e) {
      System.err.println(e);
      break; //koniec pętli for w razie nieznanego hosta
    }
    catch (IOException e) {
      // System.err.println(e);  - nie drukuj informacji o braku serwera
     }
  }

 }

}  // koniec public class SkanerPortow
```

Program powyższy jest skanerem portów na których działają aplikacje (serwery). Skanowane porty 0-1023 należą do zakresu portów, które na maszynach UNIX-owych przynależą do administratora, to znaczy tylko *root* może uruchamiać serwery na tych portach. Odwzorowanie portów na aplikację można znaleźć w pliku ustawień /etc/services. Klasa *Socket* posiada wiele metod umożliwiających między innymi uzyskanie informacji związanych z obiektem tej klasy takich jak:
- getLocalAddress() - zwraca lokalny adres, do którego dowiązane jest gniazdo,
- getLocalPort() - zwraca lokalny port, do którego dowiązane jest gniazdo,
- getInetAddress() - zwraca zdalny adres, do którego gniazdo jest podłączone,
- getPort() - zwraca zdalny numer portu, do którego gniazdo jest podłączone.

Kolejny przykład obrazuje wykorzystanie metody *getLocalPort*().

Przykład 8.7:

```
// SkanerPortow2.java:

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

public class SkanerPortow2 {

 public static void main(String[] args) {

   Socket gniazdo;
   String host = "localhost";

   if (args.length > 0) {
     host = args[0];
   }
   for (int n = 0; n < 1024; n++) {
     try {
       gniazdo = new Socket(host, n);
        int lokalPort = gniazdo.getLocalPort();
        System.out.println("Numer lokalnego portu to:" + lokalPort);
       System.out.println("Znalazłem serwer na porcie " + n + " komputera: " + host);
     }
```

```
      catch (UnknownHostException e) {
        System.err.println(e);
        break;
      }
      catch (IOException e) {
        //System.err.println(e);
       }
    }

  }
}// koniec public class SkanerPortow2
```

Powiedziano wcześniej, że idea gniazd związana była ze stworzeniem strumienia, do którego można pisać, i z którego można czytać. Podstawową, uniwersalną a zarazem abstrakcyjną klasą obsługującą strumień wejściowy jest klasa *InputStream* dla strumienia wyjściowego jest to *OutputStream*. Obydwie klasy są rozszerzane i stanowią podstawę przepływu danych (np. z i do pliku, z i do urządzenia, z i do sieci). W celu obsługi sieci w klasie *Socket* zdefiniowano metody zwracające obiekty klas *InputStream* oraz *OutputStream*, co umożliwia stworzenie strumieni do przesyłania danych przez sieć. Najczęściej wykorzystywane klasy nie abstrakcyjne to *BufferedReader* obsługujące czytanie danych przez bufor oraz *PrintStream* dla obsługi zapisu nie zwracająca wyjątku *IOException*. W celu zobrazowania pracy ze strumieniami posłużmy się przykładami. Pierwszy przykład prezentuje działanie typowe dla klienta, a więc czytanie ze strumienia.
Przykładowy program łączy się z serwerem czasu a następnie wyświetla ten czas na ekranie.

Przykład 8.8:

```
//Zegar.java:

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

public class Zegar {

  public static void main(String[] args) {

    Socket gniazdo;
    String host = "localhost";
    BufferedReader strumienCzasu;

    if (args.length > 0) {
      host = args[0];
    }

      try {
        gniazdo = new Socket(host, 13);
        strumienCzasu = new BufferedReader(new InputStreamReader(gniazdo.getInputStream()));
        String czas = strumienCzasu.readLine(); //wprowadź linię znaków z bufora strumienia
        System.out.println("Na "+host+" jest: "+czas);

      }
```

```
      catch (UnknownHostException e) {
        System.err.println(e);
      }
      catch (IOException e) {
        System.err.println(e);
       }

   }
}//koniec public class Zegar
```

Kolejny przykład umożliwi pokazanie zarówno stworzenie programu klienta jak i programu serwera.

Przykład 8.9:

```
//KlientEcho.java:

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

public class KlientEcho {

  public static void main(String[] args) {

    Socket gniazdo;
    String host = "localhost";
    BufferedReader strumienEcha, strumienWe;
    PrintStream strumienWy;
    String echo;

    if (args.length > 0) {
      host = args[0];
    }

      try {
        gniazdo = new Socket(host, 7);  //port 7 jest standardowym portem obslugi echa
         strumienWe = new BufferedReader(new InputStreamReader(gniazdo.getInputStream()));
            //czytaj z serwera
         strumienWy = new PrintStream(gniazdo.getOutputStream());
        strumienEcha = new BufferedReader(new InputStreamReader(System.in));
        //czytaj z klawiatury
         while(true){
            echo=strumienEcha.readLine();
             if (echo.equals(".")) break; //znak . oznacza koniec pracy
            strumienWy.println(echo); //wyslij do serwera
            System.out.println(strumienWe.readLine());  //wyslij na monitor
         }
      }
      catch (UnknownHostException e) {
        System.err.println(e);
      }
      catch (IOException e) {
        System.err.println(e);
       }
```

```
  }
}//koniec public class KlientEcho
```

Program serwera

Przykład 8.10:

```
//SerwerEcho.java:

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

public class SerwerEcho {

  public static void main(String[] args) {

    ServerSocket serwer;
    Socket     gniazdo;
    String host = "localhost";
    BufferedReader strumienEcha, strumienWe;
    PrintStream strumienWy;
    String echo;

    if (args.length > 0) {
      host = args[0];
    }

      try {
       serwer = new ServerSocket(7); //stworz serwer pracujacy na porcie 7 biezacego komputera
         while(true){ //główna pętla serwera
             try{
               while(true){ //główna pętla połączenia
                 gniazdo = serwer.accept();  //przyjmuj połączenia i stworz gniazdo
                 System.out.println("Jest polaczenie");
                 while(true){
                   strumienWe = new BufferedReader(new InputStreamReader(gniazdo.getInputStream()));
                   strumienWy = new PrintStream(gniazdo.getOutputStream());
                   echo=strumienWe.readLine();
                   strumienWy.println(echo); //wyslij to co przyszlo
                 }//od while

              }//od while
           }//od try
           catch (SocketException e){
               System.out.println("Zerwano polaczenie"); //klient zerwał połączenie
           }
           catch (IOException e) {
                System.err.println(e);
           }
         }//od while
      }// od try
      catch (IOException e) {
```

```
        System.err.println(e);
      }
   }
}//koniec public class SerwerEcho
```

Przykład SerwerEcho.java demonstruje zastosowanie klasy *ServerSocket* do tworzenia serwerów w architekturze klient-serwer. Klasa *ServerSocket* dostarcza kilka konstruktorów umożliwiających stworzenie serwera na danym porcie, z określoną maksymalną liczbą połączeń (kolejka), na danym porcie danego komputera o konkretnym IP (ważne w przypadku wielu interfejsów) z określoną maksymalną liczbą połączeń (kolejka). Kolejnym istotnym elementem implementacji klasy *ServerSocket* jest oczekiwanie i zamykanie połączeń. W tym celu wykorzystuje się metody *ServerSocket.accept*() oraz *Socket.close*(). Metoda *accept*() blokuje przepływ instrukcji programu i czeka na połączenie z klientem. Jeśli klient podłączy się do serwera metoda *accept*() zwraca gniazdo, na którym jest połączenie. Zamknięcie połączenia odbywa się poprzez wywołanie metody *close*() dla stworzonego gniazda. Wówczas połączenie jest zakończone i aktywny jest nasłuch *accept*() oczekujący na następne podłączenie. Jeżeli jest potrzeba zwolnienia portu serwera to wówczas należy wywołać metodę *ServerSocket.close*(). Inne przydatne metody klasy *ServerSocket* to *getLocalPort*() umożliwiająca uzyskanie numeru portu na jakim pracuje serwer oraz metoda *setSoTimeout*() umożliwiająca danemu serwerowi ustawienie czasu nasłuchu metodą *accept*(). Metoda *setSoTimeout*() musi być wywołana przed *accept*(). Czas podaje się w milisekundach. Jeżeli w zdefiniowanym okresie czasu *accept*() nie zwraca gniazda (brak połączenia) zwracany jest wyjątek *InterruptedException*.

8.2.3 Inne mechanizmy programowania sieciowego w Javie

Java dostarcza jeszcze szereg innych mechanizmów pracy w sieci. Należy wymienić chociażby pracę w grupach roboczych poprzez zastosowanie klasy *MulticastSocket* oraz możliwości wywoływania metod i programów po stronie serwera. Te ostatnie to przede wszystkim zastosowanie pakietów java.rmi czyli stosowanie Remote Method Invocation (RMI) oraz zastosowanie pakietu javax.servlet do tworzenia aplikacji wywoływanych po stronie serwera. RMI jest standardowo dostępne w dystrybucji JDK natomiast Servlety stanowią rozszerzenie Javy, a więc pakietu muszą być dodatkowo dołączone do biblioteki Javy. Praca z RMI polega na stworzeniu odpowiedniego kodu programu po stronie serwera z wskazaniem metod dostępnych dla klienta (metody deklarowane w interfejsie rozszerzającym interfejs *Remote*). Kod programu serwera musi dodatkowo zawierać instrukcje rejestrujące metody zdalne. Zanim uruchomiony zostanie kod serwera konieczne jest uruchomienie programu rejestru metod (*rmiregistry*). Po uruchomieniu rejestru i programu serwera klienci mogą korzystać z udostępnianych metod. Dostęp do nich jest możliwy poprzez wywołanie referencji do obiektu zdalnego (*lookup*). Posiadając referencję można wykonać dowolną, udostępnianą metodę zdalną. W celu korzystania z dobrodziejstw servletów z siecią WWW należy upewnić się, że dany serwer obsługuje servlety. Servlet jest praktycznie analogiem apletu, z tym że jest wykonywany na serwerze. Stworzenie servletu wymaga zastosowanie odpowiedniej konstrukcji kodu (dziedziczenie z klas umieszczonych w pakietach

javax.servlet.*; np. GenericServlet, HttpServlet; oraz przepisania określonych funkcji) oraz zastosowania właściwych wywołań w kodzie html (np. <servlet name=”Licznik” code=”Licznik”></servlet>).

Rozdział 9 Obsługa baz danych w języku Java

9.1 Obsługa baz danych w Javie - pakiet SQL
9.2 Utworzenie połączenia z bazą danych
9.3 Sterowniki
9.4 Wysłanie polecenia SQL
9.5 Rezultaty i ich przetwarzanie

9.1 Obsługa baz danych w Javie - pakiet SQL

Ze względu na tak dużą popularność baz danych oraz z uwagi na liczne mechanizmy pracy z bazami danych jakie dostarcza język Java warto omówić podstawy bibliteki Java JDBC API. JDBC jest znakiem towarowym firmy SUN (nie jest to skrót, często tłumaczony jako Java Database Connectivity) określającym interfejs języka Java przeznaczony do wykonywania poleceń SQL (Structured Query Language). Strukturalny język zapytań SQL jest uniwersalnym, znormalizowanym (Java obsługuje SQL zgodnie z normą ANSI/ISO/IEC 9075:1992, inaczej SQL-2) językiem pracy z bazami danych. W oparciu o SQL tworzone są systemy zarządzania bazami danych (DBMS - DataBase Management System), których rolą jest m.in tworzenie struktury bazy, wypełnianie bazy danych, usuwanie lub aktualizacja rekordów, nadawanie praw dostępu, itp. JDBC jest interfejsem niskiego poziomu wywołującym bezpośrednio polecenia języka SQL. Rolę JDBC można więc ująć w trzech punktach:

1. utworzenie połączenia z bazą danych,
2. wysłanie polecenia (poleceń) SQL,
3. przetworzenie otrzymanych wyników.

JDBC może stanowić podstawę do tworzenia interfejsów wyższego rzędu. Znane są prace nad stworzeniem interfejsu mieszającego elementy SQL i Javy (np. umieszczenie zmiennych Javy w SQL). Określony preprocesor wbudowanego w Javie języka SQL tłumaczyłby stworzone rozkazy na rozkazy niskiego poziomu zgodnie z JDBC. Inna wersja interfejsu wysokiego rzędu zakłada odwzorowanie tabel na klasy. Każdy rekord staje się wówczas obiektem danej klasy. Tworzenie interfejsów wyższego poziomu jest również istotne z punktu widzenia planowanego modelu dostępu do bazy danych. Popularna dwu-warstwowa metoda dostępu (two-tier) daje bezpośredni dostęp do bazy danych (aplikacja/applet - baza danych) . Oznacza to, że musimy znać format danych bazy by móc pobrać lub zmienić informacje. W przypadku bardziej uniwersalnym dodaje się trzecią warstwę w modelu dostępu do bazy (aplikacja/applet (GUI) - serwer - baza danych). W modelu trzy punktowym stosuje się interfejs wyższego poziomu po to aby struktura dostępu do bazy danych stanowiła pewną abstrakcję, co umożliwia tworzenie różnych klientów bez potrzeby zagłębiania się w szczegóły protokołów wymiany danych z bazą. Tak stworzona konstrukcja dostępu do bazy danych uwalnia klienta od znajomości organizacji bazy danych, co za tym idzie możliwe są prawie dowolne modyfikacje ustawień bazy danych np. kilka rozproszonych baz zamiast jednej.
Istnieją inne interfejsy dostępu do baz danych jak na przykład popularny Open DataBase Connectivity firmy Microsoft. Jednak korzystanie z ODBC przez programy Javy nie stanowi dobrego rozwiązania ponieważ:

1. występuje różnica języków programowania - ODBC jest stworzony w C, a co za tym idzie konieczna konwersja porzuca cechy języka Java, a często staje się nie realizowalna ze względu na inne koncepcje np. problem wskaźników
2. korzystanie z ODBC jest znacznie trudniejsze, a nauka pochłania zbyt dużo czasu,
3. praca z ODBC wymaga ręcznych ustawień na wszystkich platformach klientów, podczas gdy korzystanie z JDBC umożliwia automatyczne wykorzystanie kodu JDBC na wszystkich platformach Javy począwszy od komputerów sieciowych do superkomputerów.
W okresie wprowadzania języka Java oraz sterowników JDBC stworzono (JavaSoft) mosty JDBC-ODBC, będące rozwiązaniem dla tych, którzy korzystają z baz danych nie posiadających innych, "czystych" sterowników JDBC.

9.2 Utworzenie połączenia z bazą danych

Stworzenie połączenia z bazą danych polega utworzeniu obiektu Connection. W tym celu stosuje się jedną ze statycznych metod DriverManager.getConnection(). Każda metoda getConnection() zawiera jako argument adres URL dostępu do bazy danych. Adres ten definiowany jest poprzez trzy człony:

jdbc:<subprotocol>:<subname>.

Pierwszy element adresu jest stały i nosi nazwę jdbc. Określa on typ protokołu. Kolejny element stanowi nazwa sterownika lub mechanizmu połączenia do bazy danych. Przykładowo mogą to być nazwy: msql - sterownik dla bazy mSQL, odbc - mechanizm dla sterowników typu ODBC. Ostatna część adresu zawiera opis konieczny do zlokalizowania bazy danych. Element ten zależy od sterownika czy mechanizmu połączeń i może zawierać dodatkowe rozszerzenia zgodnie z koncepcją przyjętą przez twórcę sterownika. Standardowo omawiana część adresu wygląda następująco:

//hostname:port/subsubname.

Przykładowe pełne adresy url mogą wyglądać następująco:

jdbc:odbc:biomed, jdbc:msql://athens.imaginary.com:4333/db_test.

Jedna z metod getConnection() umożliwia dodatkowo przesłanie nazwy użytkownika i hasła dostępu do bazy danych:

getConnection(String url, String user, String password).

Połączenie byłoby niemożliwe bez istnienia sterowników. Zarządzaniem sterownikami, które zarejestrowały się za pomocą metody DriverManager.registerDriver() zajmuje się klasa DriverManager (np. metody getDriver(), getDrivers()). Klasy sterowników powinny zawierać kod statyczny (static {}), który w wyniku ładowania tych klas stwarza obiekt danej klasy automatycznie rejestrującej się za pomocą metody DriverManager.registerDriver(). Ładowanie sterownika (a więc jego rejestracja) odbywa się najczęściej poprzez wykorzystanie

metody Class.forName(). Ta metoda ładowania sterownika nie zależy od ustawień zewnętrznych (konfiguracji sterowników) i ma charakter dynamiczny. Przykładowe ładowanie sterownika o nazwie "oracle.db.Driver" wykonane jest poprzez zastosowanie metody Class.forName("oracle.db.Driver"). Fragment kodu obrazujący etap łączenia się z bazą danych ukazano poniżej:

```
 String url = "jdbc:odbc:kurs";
// przykładowa nazwa drivera - slowo "kurs" jest nazwą zasobów
//definiowaną w ODBC dla bazy np. pliku tekstowego
String username = ""; //brak parametrów dostępu do pliku tekstowego
String password = "";
try {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   //ładowanie sterownika -  most JDBC-ODBC
} catch (Exception e) {
   System.out.println("Blad ladowania sterownika JDBC/ODBC.");
   return;
}
Connection c = null;
try {
   c = DriverManager.getConnection (url, username, password);  //połączenie
} catch (Exception e) {
   System.err.println("Wystapil problem z polaczeniem do "+url);
}
```

Powyższy przykład zakłada rejestrację w ODBC bazy tekstowej o dostępie "kurs" i wskazanie odpowiedniego katalogu.

9.3 Sterowniki

Sterowniki, a więc pakiety kodu zawierającego implementację deklarowanych klas i metod (obsługa dostępu do bazy, np. implementacja metod interfejsu ResultSet jak first() getInt(); itp.) są zazwyczaj dzielone na cztery grupy:

a. sterowniki JDBC odwzorowujące żądaną funkcjonalność na funkcjonalność sterowników Open Database Connectivity – ODBC. Sterowniki te oznaczanę są często jako mosty JDBC-ODBC. Sun standardowo dostarcza swoją wersję mostu.
b. Sterowniki JDBC odwzorowujące żądaną funkcjonalność na funkcjonalność dostępną poprzez sterowniki binarne danej bazy danych.
c. Sterowniki JDBC odwzorowujące żądaną funkcjonalność na funkcjonalność oprogramowania pośredniczącego w komunikacji z serwerem bazy danych, czyli z następuje tłumaczenie poleceń.
d. Sterowniki JDBC odwzorowujące żądaną funkcjonalność na funkcjonalność serwera bazy danych. Jest to w pełni zgodne z Javą rozwiązanie, które zapewnia najczęściej twórca oprogramowania serwera bazy danych.

Poniższy fragment kodu demonstruje zasady tworzenia połączenia i wykorzystywania kodu zdalnego (sterowniki) dla bazy danych „qabase” pracującej na serwerze Msql:

```
try{
```

```
        Class.forName("com.imaginary.sql.msql.MsqlDriver");
}catch (Exception e){
        System.out.println("Błąd wczytywania sterowników");
        return;
}
String URL = "jdbc:msql://biomed.eti.pg.gda.pl:1114/qabase";
String username ="msql";
String password="";
s=null;
con=null;
try{
        con=DriverManager.getConnection(URL,username,password);
        s=con.createStatement();
}catch (Exception e) {
        System.err.println("Błąd połączenia z "+URL);
}
```

W przypadku apletu sterowniki (pakiet kodu) musi być zainstalowany na serwerze WWW tam, skąd pochodzi aplet.

9.4 Wysłanie polecenia SQL

W celu wysłania polecenia SQL należy stworzyć obiekt Statement. Obiekt ten stanowi kontener dla wykonywanych poleceń SQL. Wykorzystywane są dodatkowo dwa kontenery: PreparedStatement oraz CallableStatement. Obiekt Statement jest wykorzystywany do wysyłania prostych poleceń SQL nie zawierających parametrów, obiekt PreparedStatement używany jest do wykonywania prekompilowanych (przygotowanych - prepared) poleceń SQL zawierających jedno lub więcej pól parametrów (oznaczanych znakiem "?"; tzw. parametry IN), natomiast obiekt CallableStatement jest wykorzystywany do stworzenia odwołania (call) do przechowywanych w bazie danych procedur. W celu stworzenia obiektu dla opisanych wyżej interfejsów wykorzystuje się trzy odpowiednie metody interfejsu Connection: createStatement() - dla interfejsu Statement, prepareStatement() - dla interfejsu PreparedStatement oraz prepareCall() dla interfejsu CallableStatement. Przykładowo fragment kodu tworzący obiekt wyrażenia Statement może wyglądać następująco:

```
Connection c = null;
    try {
       c = DriverManager.getConnection (url, username, password);  //połączenie
       Statement s = c.createStatement(); // tworzymy obiekt wyrażenia
    } catch (Exception e) {
       System.err.println("Wystapil problem z polaczeniem do "+url);
    }
```

Posiadając obiekt Statement można wykorzystać trzy podstawowe metody umożliwiające wykonanie polecenia SQL. Pierwsza z metod executeQuery() jest używana do wykonywania poleceń, których efekt daje pojedynczy zbiór rezultatów ResultSet np. wyrażenie SELECT - wybierz. Drugą metodę executeUpdate() wykorzystuje się przy wykonywaniu poleceń INSERT, UPDATE oraz DELETE a

także wyrażeń typu SQL DDL (Data Definition Language - język definicji danych) jak CREATE TABLE i DROP TABLE. Efekt działania pierwszych trzech poleceń daje modyfikację jednej lub więcej kolumn w zero i więcej wierszach tabeli. Zwracana wartość w wyniku działania metody executeUpdate() to liczba całkowita wskazująca ilość rzędów, które podlegały modyfikacjom. Dla wyrażeń SQL DDL zwracana wartość jest zawsze zero. Metoda execute() jest rzadko używana, ponieważ jest przygotowana do obsługi poleceń zwracających więcej niż jeden zbiór danych (więcej niż jeden obiekt ResultSet). Obsługa zwracanych danych jest więc kłopotliwa stąd metoda ta jest wykorzystywana dla obsługi specjalnych operacji. W przypadku obiektu PreparedStatement interfejs definiuje własne metody execute(), executeUpdate() oraz executeQuery(). Dlaczego? Czy nie wystarczy, że interfejs PreparedStatement dziedziczy wszystkie metody interfejsu Statement, a więc i te omawiane. Otóż nie. Obiekty Statement nie zawierają wyrażenia SQL, które musi być podane jako argument do ich metod. W przypadku obiektu PreparedStatement wyrażenie SQL musi być przygotowane – obiekt zawiera prekompilowane wyrażenie SQL. Dlatego wykonanie odpowiedniego polecenia polega na wywołaniu metody dla obiektu PreparedStatement bez podawania żadnego argumentu.
Poniżej przedstawiono porównanie wykonania polecenia SQL dla obiektu Statement oraz obiektu PreparedStatement:

Statement:
```
static String SQL = "INSERT INTO kurs VALUES ('Mariusz', 'MK', 28)";
Statement s=...
s.executeUpdate(SQL);
```

PreparedStatement:
```
Connection c=...
PreparedStatement ps = c.prepareStatement("INSERT INTO kurs VALUES (?, ?, ?)");
        ps.setString(1,"Mariusz");
        ps.setString(2,"MK");
        ps.setInt(1, 28);
        ps.executeUpdate();
```

W pracy z zewnętrznymi procedurami przechowywanymi poza kodem programu istotą wykorzystania Javy jest stworzenie wyrażenia typu CallableStatement poprzez podanie jako parametru metody Connection.prepareCall() sekwencji ucieczki typu {call procedure_name[(?, ?)]} lub {? = call procedure_name[(?, ?)]} w przypadku gdy procedura zwraca wartość. Jak widać istota polega na znajomości nazwy i parametrów obsługiwanej procedury. Parametry ustawia się tak jak dla wyrażeń PreparedStatement natomiast pobiera się metodami getXXX(). Wykonanie polecenia (poleceń) procedury odbywa się poprzez wykorzystanie metody execute().

Posiadając wiedzę na temat stworzenia połączenia oraz przygotowywania i wykonywania wyrażeń można wykonać dwa przykładowe programy. Programy te wymagają ustawienia w ODBC systemu Win95/NT (Panel sterowania): dodanie ODBC typu plik tekstowy, nazwa udziału: kurs, katalog: c:\kurs\java. Ustawienia te są konieczne ponieważ wykorzystany zostanie pomost JDBC-ODBC jako sterownik do bazy danych. Obydwa programy generują to samo: bazę danych (pliki) o nazwie kurs

z przykładowymi danymi. Pierwszy program czyni to z pomocą wyrażeń Statement drugi z pomocą PreparedStatement.

Przykład 8.1:

```
//DBkurs.java:
import java.sql.*;

public class DBkurs {
  static String[] SQL = {
    "CREATE TABLE kurs ("+
      "uczestnik varchar (32),"+
      "inicjaly varchar (3),"+
      "wiek integer)",
    "INSERT INTO kurs VALUES ('Jarek', 'JS', 30)",
    "INSERT INTO kurs VALUES ('Andrzej', 'AM', 27)",
    "INSERT INTO kurs VALUES ('Ania', 'AM', 20)",
    "INSERT INTO kurs VALUES ('Marcin', 'MH', 25)",
  };
  public static void main(String[] args) {
    String url = "jdbc:odbc:kurs";
    String username = "";
    String password = "";
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (Exception e) {
      System.out.println("Blad ladowania sterownika JDBC/ODBC.");
      return;
    }
    Statement s = null;
    Connection c = null;
    try {
      c = DriverManager.getConnection (url, username, password);
      s = c.createStatement();
    } catch (Exception e) {
      System.err.println("Wystapil problem z polaczeniem do "+url);
    }
    try {
      for (int i=0; i<SQL.length; i++) {
        s.executeUpdate(SQL[i]);
      }
      c.close();
    } catch (Exception e) {
      System.err.println("Wystapil problem z wyslaniem SQL do "+url+
        ": "+e.getMessage());
    }
  }
}// koniec public class DBkurs
```

Przykład 8.2:

```
//DBkurs2.java:
import java.sql.*;

public class DBkurs2 {
  static String SQL = "CREATE TABLE kurs ("+
```

```
        "uczestnik varchar (32),"+
        "inicjaly varchar (3),"+
        "wiek integer)";
  public static void main(String[] args) {
    String url = "jdbc:odbc:kurs";
    String username = "";
    String password = "";
     String[] imie= {"Jarek","Andrzej","Ania","Marcin"};
     String[] inic={"JS","AM", "AM", "MH"};
     int[] lat={30,27,20,25};
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (Exception e) {
      System.out.println("Blad ladowania sterownika JDBC/ODBC.");
      return;
    }
    Connection c = null;
     Statement s = null;
     PreparedStatement ps = null;
    try {
      c = DriverManager.getConnection (url,username,password);
      s = c.createStatement();
    } catch (Exception e) {
      System.err.println("Wystapil problem z polaczeniem do "+url);
    }
    try {
       s.executeUpdate(SQL);
       s.close();
        ps = c.prepareStatement("INSERT INTO kurs VALUES (?, ? , ?)");
        for(int n=0; n<4; n++){
            ps.setObject(1,imie[n]);
          //setObject zamiast setString z uwagi na problemy z konwersją typów danych do Text File ODBC (MS)
            ps.setObject(2,inic[n]);
            ps.setInt(3, lat[n]);
            ps.executeUpdate();
        }
       ps.close();
    } catch (Exception e) {
      System.err.println("Wystapil problem z wyslaniem SQL do "+url+ ": "+e.getMessage());
    }
    finally{
     try{ c.close(); }
     catch(SQLException e) {
        e.printStackTrace();
     }
    }//od finally
  }
}// koniec public class DBkurs2
```

Dla potrzeb pracy z bazą danych za pomocą języka SQL istotne jest odwzorowanie typów danych Java->JDBC->SQL->DBMS. JDBC 2.0 API jest zgodny z SQL-2, a co więcej udostępnia typy danych zgodnych z propozycją standardu SQL-3 (np. typ danych BLOB). Teoretycznie twórca oprogramowania bazy danych powinien traktować typy danych z baz danych tak jakby to były typy danych SQL (i tak powinno

być). Oznacza to ponownie, że baza danych stanowi pewną abstrakcję. W pracy z wyrażeniami w JDBC istotne jest ustawianie parametrów wyjściowych, co powoduje wykorzystanie jednej z metod setXXX(); a także ważne jest ustawianie parametrów wejściowych (CallableStatement, ResultSet), co powoduje wykorzystanie jednej z metod getXXX(). W metodach pracy z danymi XXX oznacza typ danych. Poniższa tabela ukazuje przykładowe odwzorowanie typów pomiędzy Javą a JDBC(SQL) poprzez użycie metod getXXX().

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x
getBytes														X	X	x			
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimestamp											x	x	x				x		X
getAsciiStream											x	x	X	x	x	x			
getUnicodeStream											x	x	X	x	x	x			
getBinaryStream														x	x	X			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Rysunek 9.1 : Konwersja typów danych

Przy pracy z wyrażeniami SQL (a właściwie wykonywaniem tych wyrażeń) ważnym zagadnieniem są transakcje. Transakcja składa się z jednego lub więcej poleceń, które zostały wywołane, wykonane i potwierdzone (Commit) lub odrzucone (RollBack). Transakcja stanowi więc jednostkową operację. Zarządzanie transakcją

jest szczególnie ważne gdy chcemy wykonać naraz kilka operacji. Wówczas konieczne jest ustawienie pracy w stanie bez potwierdzania (Connection.setAutoCommit(false)- domyślnie ustawiona jest praca z automatycznym potwierdzaniem), a następnie po wykonaniu wyrażeń SQL wydawany jest rozkaz potwierdzenia (Connection.commit()). Potwierdzenie jest poleceniem, które powoduje, że zmiany spowodowane wyrażeniem SQL stają się stałe. W przypadku błędu (np. spowodowanego awarią sieci komputerowej) można wysłać polecenie przerwania (RollBack) powodujące odtworzenie stanu przed wykonaniem wyrażenia. Podsumowując można powiedzieć, że efekt działania każdego polecenia SQL jest tymczasowy do momentu wysłania polecenia potwierdzenia, kiedy to zmiany stają się stałe lub do momentu wysłania polecenia przerwania, kiedy to odtwarzany jest stan przed wykonaniem polecenia.

9.5 Rezultaty i ich przetwarzanie

W zależności od typu polecenia SQL, a co za tym idzie typu wyrażenia executeXXX(), możliwe są różne rezultaty wykonanych operacji. Metoda executeUpdate() zwraca liczbę zmienionych wierszy, metoda executeQuery() zwraca obiekt typu ResultSet zawierający wszystkie rekordy (wiersze) będące wynikiem wydania polecenia SQL (SELECT). Dostęp do danych zawartych w ResultSet następuje poprzez odpowiedni przesuw (ResultSet.next()) po rekordach (początkowo wskaźnik ustawiony jest przed pierwszym elementem) oraz odczyt wartości pól za pomocą metod podanych w powyższej tabelce typu getXXX(), gdzie identyfikator kolumny określany jest poprzez jej nazwę (typu String - ważna jest wielkość liter) lub numer kolejny w rekordzie (np. 1 kolumna, 2 kolumna, ...).
Informację o kolumnach obiektu ResultSet dostępne są poprzez wywołanie metod interfejsu ResultSetMetaData, którego obiekt zwracany jest poprzez przywołanie metody ResultSet.getMetaData. Przykładowe własności kolumn to nazwa kolumny- getColumnName(), czy typ kolumny- getColumnType(). Niektóre systemy zarządzania bazami danych umożliwiają tzw. pozycjonowane zmiany i kasowanie polegające na wprowadzaniu zmian w aktualnie dostępnym rekordzie w ResultSet. Możliwe jest wówczas wykorzystanie metod updateXXX() interfejsu ResultSet do aktualizacji danych. W celu zobrazowania techniki przetwarzania rezultatów wykonania polecenia SQL posłużmy się następującym przykładem:

Przykład 8.3:

```
//DBkurs3.java:

import java.sql.*;

public class DBkurs3 {


   public static void main(String[] args) {
      String url = "jdbc:odbc:kurs";
      String username = "";
      String password = "";
       String imie, inic;
       int lata;
```

```
        try {
           Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch (Exception e) {
           System.out.println("Blad ladowania sterownika JDBC/ODBC.");
           return;
        }

        Connection c = null;
         Statement s = null;
        try {
           c = DriverManager.getConnection (url,username,password);
           s = c.createStatement();
        } catch (Exception e) {
           System.err.println("Wystapil problem z polaczeniem do "+url);
        }
         try {
              ResultSet r=s.executeQuery("SELECT uczestnik, inicjaly, wiek FROM kurs");
              int n=0;
              while(r.next()){
                    n++;
                    imie=r.getString(1);  // w pętli pobieramy dane
                    inic=r.getString(2);
                    lata=r.getInt(3);
                    System.out.println("Dane rekordu nr: "+n+" to: "+imie+", "+inic+", "+lata);
              }
              s.close();
        } catch (Exception e) {
           System.err.println("Wystapil problem z wyslaniem SQL do "+url+ ": "+e.getMessage());
        }
        finally{
          try{ c.close(); }
          catch(SQLException e) {
              e.printStackTrace();
          }

         }//od finally
   }
}// koniec public class DBkurs3
```

W ten sposób pokazane zostały wszystkie trzy podstawowe zadania w pracy z bazami danych. Warto wspomnieć, że JDBC składa się faktycznie z dwóch API: jedna biblioteka standardowa dołączana do JDK (i tylko te elementy wykorzystano i omówiono powyżej) tzw. JDBC 2.0 core API oraz dodatkowa biblioteka będąca rozszerzeniem czyli JDBC 2.0 extension API. Tak ujęta budowa JDBC umożliwia ciągły rozwój biblioteki przy zachowaniu standardowych narzędzi dostarczanych z podstawowym środowiskiem Javy.

Rozdział 10 Bezpieczeństwo w Javie

10.1 Bezpieczeństwo programów i danych
10.2 Bezpieczeństwo w Javie
10.3 Obsługa zasad bezpieczeństwa w Javie
10.4 Kryptografia
10.4.1 Kryptografia w Javie
10.4.2 Skróty wiadomości
10.4.3 Kod autentyczności wiadomości - MAC
10.4.4 Klucze i podpis cyfrowy
10.4.5 Kodowanie danych

10.1 Bezpieczeństwo programów i danych

Termin "bezpieczeństwo" ("security") można przykładowo opisać poprzez funkcjonalność programów, które powinny być:
- wolne od wrogich podprogramów - program nie powinien utrudniać pracy w danym środowisku oraz powinien być wolny od niezamierzonych podprogramów np. wirusów;
- "nieinwazyjne" - programy nie powinny mieć dostępu do informacji prywatnych przechowywanych na komputerze lokalnym lub w sieci (kontrolowany dostęp do zasobów prywatnych),
- autoryzowane - programy powinny umożliwiać ustalenie i potwierdzenie tożsamości autora i danych,
- kodowane - programy powinny umożliwiać kodowanie danych,
- kontrolowane - programy powinny umożliwiać tworzenie rejestrów operacji związanych z zagrożeniem bezpieczeństwa (np. pliki typu *.log),
- sterowane - programy powinny być zabezpieczone przed wykorzystywaniem zbyt dużej ilości zasobów,
- certyfikowane - programy powinny spełniać wymagania i uzyskiwać certyfikaty bezpieczeństwa np. C2 czy B1 według systemu opracowanego dla rządu USA i innych.

10.2 Bezpieczeństwo w Javie

Środowisko Java 1.0 udostępniało pierwsze dwa elementy. Kolejne wersje Javy wprowadzały dodatkową funkcjonalność programów i tak JAVA 2 umożliwia spełnienie funkcjonalności zgodnie z pierwszymi pięcioma punktami, z tym, że kodowanie danych dostępne jest tylko dla obywateli USA i Kanady (ograniczenia eksportowe) poprzez rozszerzenie JCE (Java Cryptography Engine). Model bezpieczeństwa tworzony dla aplikacji Javy ewaluował poprzez kolejne wersję platformy. Począwszy od wersji 1.0 opracowano model, który później nazwano "piaskownicą". Odwołanie się do piaskownicy ukazuje analogię programu Javy do dziecka bawiącego się w piaskownicy. Dziecko w piaskownicy ma ograniczony dostęp do otoczenia (wyznaczany przez rozmiar piaskownicy) oraz ma do dyspozycji określone zabawki (zasoby). Tak samo program Javy umieszczony w danym systemie bezpieczeństwa ma takie możliwości jakie daje mu ten system. W wersji 1.0 programy Javy, lub bardziej ogólnie kody dzielone są na zaufane i niezaufane. Kod zaufany to ten, który znajduje się w lokalnym systemie; kod niezaufany to ten, który

pochodzi ze zdalnych urządzeń (np. z sieci). Kod zaufany ma pełny dostęp (pełny w sensie danego systemu operacyjnego - jeśli np. plik nie jest do odczytu przez danego użytkownika w danym systemie operacyjnym, to plik ten będzie również niedostępny przez kod Javy) do zasobów, kod niezaufany ma dostęp taki, na jaki zezwala mu "piaskownica". Standardowo wszystkie applety uruchamiane są w obrębie jakiejś aplikacji np. Netscape, HotJava, appletviewer; które tworzą "piaskownicę" dla danego appletu. W wersji Java1.0 wszystkie applety nie mają dostępu do zasobów lokalnych, co jest uwarunkowane przez model bezpieczeństwa tej platformy Javy. Jako ciekawostkę można podać, że przyczyną dla której nie tworzono dostępu dla appletów do lokalnych zasobów plików nie był aspekt bezpieczeństwa, lecz fakt, że docelowo w propagowanej przez Suna technologii komputerów sieciowych takich zasobów po prostu być nie powinno. Dla aplikacji uruchamianych lokalnie można również stworzyć programy, które będą ustalać dla nich "piaskownicę". Wówczas docelowa aplikacja będzie wywoływana jako atrybut programu ustalającego bezpieczeństwo. Można więc zarówno zdalnie jak i lokalnie ustalić prawa dostępu do zasobów plików, co jest niezwykle przydatne, szczególnie dla programów pracujących z danymi chronionymi (np. w medycynie). W wersji JDK 1.1 wprowadzono zmianę w modelu bezpieczeństwa polegającą na tym, że tzw. podpisany elektronicznie kod (aplet) jest traktowany jako kod zaufany, a co za tym idzie ma dostęp do lokalnych zasobów komputera. Podpisane kody wraz z kluczami są dostarczane do systemu w pliku spakowanym JAR. Platforma Java 2 wprowadza liczne zmiany związane z bezpieczeństwem. Nowy model bezpieczeństwa przyjmowany przez tą platformę zakłada, że każdy kod (lokalny i zdalny, podpisany czy nie) podlega kontroli zasad bezpieczeństwa (policy). W Javie 2 nie istnieje już wbudowany mechanizm dzielący kod na zaufany i niezaufany. Każdy kod podlega kontroli względem ustawień zasad bezpieczeństwa, które to zasady są spisywane w plikach konfiguracyjnych JRE lub definiowanych przez użytkownika plikach zasad bezpieczeństwa. Po kontroli zasad bezpieczeństwa dany kod uzyskuje określone uprawnienia wynikające z ustawień zasad bezpieczeństwa. Przykładowo aplet A podpisany prze X może mieć dostęp do plików w lokalnym systemie plików, dający mu prawo zapisu pliku o nazwie: pliktest. Domyślnie przyjęto, że zasady bezpieczeństwa dają uprawnienia programom tak jak to było w modelu 1.0, tzn. zasady bezpieczeństwa dla apletów uniemożliwiają im dostęp do lokalnego zasobu plików. Określone jest to poprzez fakt, że dla aplikacji nie instalowany jest automatycznie system zarządzania bezpieczeństwem, tak jak to ma miejsce dla apletów. W celu uruchomienia aplikacji z działającym systemem zarządzania bezpieczeństwem należy wywołać interpreter (java) z opcją -Djava.security.manager. Wówczas aplikacje podlegają takiej samej kontroli bezpieczeństwa jak aplety poprzez pliki konfiguracyjne ustawień bezpieczeństwa.
Java 2 wprowadza jako rozszerzenia pojęcia "piaskownica" pojęciem "dziedzina". Otóż w czasie wykonywania kodu, kody grupowane są w grupy kodów (pochodzących z tego samego źródła - CodeBase) o tych samych prawach dostępu. Grupy te zwane dalej dziedzinami oznaczają określony zakres dostępu do zasobów (według ustalonych zasad bezpieczeństwa). Ustalając więc zakres bezpieczeństwa można uzyskać informację (obiekt klasy PermissionCollection) wywołując metodę getPermissions() klasy Policy.

Podsumowując można wskazać następujące elementy związane z ustalaniem zakresu bezpieczeństwa dla aplikacji Javy:

- weryfikacja kodu: weryfikator B-kodu sprawdza czy klasy Javy spełniają prawa języka Java (klasy zestawu core API nie są sprawdzane);
- ładowanie klas: jeden lub więcej systemów ładowania klas wprowadza wszystkie klasy, które nie są częścią core API (do wersji Java 2, ładowanie klas wprowadzało klasy których nie było w ustawieniu CLASSPATH);
- kontrola dostępu: od Javy 2 kontroler dostępu umożliwia lub zabrania dostępu do systemu operacyjnego;
- zarządzanie bezpieczeństwem: system zarządzania bezpieczeństwem był szczególnie istotny w wersjach wcześniejszych niż 2, ponieważ zarządzał on dostępem do zasobów komputera. Od wersji Java 2 system zarządzania bezpieczeństwem odwołuje się bezpośrednio do kontrolera dostępu;
- pakiet java.security: umożliwia dostęp do powyższych systemów (poprzez szereg klas i interfejsów np. AccessController, SecurClassLoader) oraz dostarcza szeregu narzędzi do tworzenia podpisów cyfrowych, kluczy, certyfikatów;
- baza danych kluczy: zbiór kluczy używanych przez system zarządzania bezpieczeństwem i kontrolera dostępu do sprawdzania podpisów elektronicznych związanych z podpisanymi plikami klas Javy.

W Javie 2 należy w sposób szczególny omówić kontroler dostępu ponieważ większość zagadnień bezpieczeństwa jest z nim związana. Kontroler dostępu (AccessController) jest zbudowany w oparciu o cztery zasadnicze elementy:
- CodeSource; reprezentacja źródła pochodzenia kodu;
- Permissions; reprezentacja uprawnień dostępu;
- Policies; reprezentacja wszystkich uprawnień, które powinny być nadane dla danego kodu - określenie zasad bezpieczeństwa;
- Protection Domains; reprezentacja konkretnego źródła i wszystkich uprawnień mu nadanych.

10.3 Obsługa zasad bezpieczeństwa w Javie

Zasady bezpieczeństwa dla platformy Javy są spisane w plikach konfiguracyjnych. Podstawowym plikiem konfiguracyjnym jest plik java.security umieszczony standardowo w katalogu <JRE_home>/lib/security. Plik ten zawiera informacje o lokalizacji pliku zasad bezpieczeństwa, o możliwości tworzenia własnych (w katalogu domowym) plików zasad bezpieczeństwa, o możliwości uruchamiania JVM z dodatkowym plikiem bezpieczeństwa (-Djava.security.policy=własnyplikbezpieczeństwa), o typie magazynu kluczy oraz o dostawcy i lokalizacji pakietu mechanizmów bezpieczeństwa (np. algorytmy kodowania wiadomości, algorytmy generacji podpisów elektronicznych, itp.). Lokalizacja plików zasad bezpieczeństwa w pliku java.security jest ustalana poprzez podanie linii typu policy.url.n=URL; gdzie n jest kolejnym plikem zasad, a URL adresem URL do tego pliku. Pierwszym plikiem zasad bezpieczeństwa jest plik java.policy znajdujący się w tym samym katalogu co java.security. Ponieważ jest to pierwszy, domyślny plik zasad to w pliku java.security ustawiona jest następująca linia konfiguracji: policy.url.1=file:${java.home}/lib/security/java.policy. Jako drugi plik zasad bezpieczeństwa podaje się plik ustawień dla aktualnego użytkownika (z jego katalogu domowego) policy.url.2=file:${user.home}/.java.policy. Dalej można tworzyć dodatkowe pliki zasad bezpieczeństwa, które mogą być dodawane według omówionej zasady do pliku java.security. Podstawowy plik zasad bezpieczeństwa

(oraz inne) zawiera wskazanie na magazyn kluczy (koniecznych przy potwierdzaniu podpisanych kodów) oraz szereg pozycji zawierających CodeSource (źródło kodu), identyfikator podpisu (jeśli taki jest użyty), oraz zestaw uprawnień (Permissions). Poniżej zaprezentowano zawartość pliku standardowego java.policy:

```
// Standard extensions get all permissions by default

grant codeBase "file:${java.home}/lib/ext/-" {
 permission java.security.AllPermission;
};

// default permissions granted to all domains

grant {
 // Allows any thread to stop itself using the java.lang.Thread.stop()
 // method that takes no argument.
 // Note that this permission is granted by default only to remain
 // backwards compatible.
 // It is strongly recommended that you either remove this permission
 // from this policy file or further restrict it to code sources
 // that you specify, because Thread.stop() is potentially unsafe.
 // See "http://java.sun.com/notes" for more information.
 permission java.lang.RuntimePermission "stopThread";

 // allows anyone to listen on un-privileged ports
 permission java.net.SocketPermission "localhost:1024-", "listen";

 // "standard" properies that can be read by anyone

 permission java.util.PropertyPermission "java.version", "read";
 permission java.util.PropertyPermission "java.vendor", "read";
 permission java.util.PropertyPermission "java.vendor.url", "read";
 permission java.util.PropertyPermission "java.class.version", "read";
 permission java.util.PropertyPermission "os.name", "read";
 permission java.util.PropertyPermission "os.version", "read";
 permission java.util.PropertyPermission "os.arch", "read";
 permission java.util.PropertyPermission "file.separator", "read";
 permission java.util.PropertyPermission "path.separator", "read";
 permission java.util.PropertyPermission "line.separator", "read";

 permission java.util.PropertyPermission "java.specification.version", "read";
 permission java.util.PropertyPermission "java.specification.vendor", "read";
 permission java.util.PropertyPermission "java.specification.name", "read";

 permission java.util.PropertyPermission "java.vm.specification.version", "read";
 permission java.util.PropertyPermission "java.vm.specification.vendor", "read";
 permission java.util.PropertyPermission "java.vm.specification.name", "read";
 permission java.util.PropertyPermission "java.vm.version", "read";
 permission java.util.PropertyPermission "java.vm.vendor", "read";
 permission java.util.PropertyPermission "java.vm.name", "read";
};
```

Ustawienia plików zasad bezpieczeństwa odbywają się poprzez edycję tekstowych plików za pomocą edytora lub poprzez standardowe narzędzie dostarczane z Java 2

tzn. policytool. Policytool jest aplikacją JAVY umożliwiającą tworzenie i edycję plików zasad bezpieczeństwa. Zanim zaprezentowane zostaną przykłady należy odnieść się do wymienionych wcześniej elementów kontrolera dostępu. W świetle znanych już metod ustalania zasad bezpieczeństwa w plikach konfiguracyjnych łatwo wytłumaczyć znaczenie czterech wymienionych elementów kontrolera dostępu. Po pierwsze obiekt klasy CodeBase reprezentuje kod (źródło), który jest lub będzie związany z określonymi uprawnieniami. Po drugie dla abstrakcyjnej klasy Permission istnieją końcowe klasy oznaczające konkretne uprawnienia. Te końcowe klasy uprawnień charakteryzują się trzema cechami: typem - oznaczającym w nazwie klasy typ uprawnień z nią związanych np. FilePermission, SocketPermission; nazwą - określającą cel z którym związane są dane uprawnienia (element String oznaczający URL, plik, itp.); akcją – określającą rodzaj uprawnień dla danego typu (np. dla FilePermission: read, write, execute,delet; dla SocketPermission: accept, listen, connect, resolve). Obiekt klasy końcowej np. FilePermission nie nadaje uprawnień w systemie operacyjnym (np. plikowi nie nadawane są uprawnienia w systemie operacyjnym lecz w środowisku Javy). Klasy końcowych uprawnień (oprócz BasicPermission-przydatna przy implementacji własnych typów uprawnień- i SecurityPermission) znajdują się poza pakietem java.security i są umieszczane w pakietach związanych z typem uprawnień np. FilePermission jest w pakiecie java.io; SocketPermission jest w pakiecie java.security, itd. Ponieważ elementarne uprawnienia mogą być mnogie dla danego typu uprawnień (np. ustawiany jest dostęp do odczytu plików z katalogu /tmp oraz drugie uprawnienie dostęp do odczytu plików z katalogu /java/test) stworzono klasę PermissionCollection, której obiekty reprezentują zbiór uprawnień.
Obiekty klasy Permissions reprezentują natomiast kolekcję obiektów klasy PermissionCollection. W ten sposób można pogrupować różne uprawnienia, które później zostaną przydzielone konkretnemu kodowi w celu stworzenia dziedziny uprawnień. Po trzecie klasa Policy ustala odwzorowanie pomiędzy kodem źródłowym a uprawnieniami. Obiekt klasy Policy jest inicjowany głównie poprzez interpretacje plików zasad bezpieczeństwa (java.policy). Zasady bezpieczeństwa są więc ustalane zewnętrznie względem kodu. Po czwarte przydzielenie konkretnemu kodowi CodeSource grupy uprawnień Permissions powoduje stworzenie dziedziny uprawnień (obiekt klasy ProtectionDomain). Obiekt klasy ProtectionDomain reprezentuje więc scaloną część w pliku konfiguracyjnym zasad bezpieczeństwa wydzieloną poprzez fragment: grant CodeBase SignedBy {Permissions}(przykładowo w powyższym pliku java.policy).
Omawiając elementy związane z uprawnieniami w Javie warto podkreślić elastyczność określania celu działania danego uprawnienia. Cel działania danego uprawnienia np. FilePermission może być ustalony poprzez podanie konkretnego elementu lub poprzez wykorzystanie różnych znaków zastępczych. Przykładowo dla uprawnień FilePermission możliwe są następujące kody sterujące:

file (oznacza konkretną nazwę pliku w bieżącym katalogu);
directory (oznacza konkretną nazwę katalogu);
directory/file (oznacza konkretną nazwę pliku w danym katalogu);
directory/* (oznacza wszystkie pliki w danym katalogu);
* (oznacza wszystkie pliki w bieżącym katalogu);
directory/- (oznacza wszystkie pliki w danym katalogu i jego podkatalogach);
- (oznacza wszystkie pliki w bieżącym katalogu i jego podkatalogach);
"<<ALL FILES>>" (oznacza wszystkie pliki w systemie plików).

Podobnie ustalane są różne znaki sterujące dla innych uprawnień. Widać stąd, że sterowanie uprawnieniami jest bardzo elastyczne. Implikację ustawiania uprawnień (np. dla pliku /tmp/jacek/test.java gdy ustalone są uprawnienia dla /tmp/-) powodują implementacje metody abstrakcyjnej klasy Permission w odpowiednich końcowych klasach uprawnień np. FilePermission.
Ustalając zasady nadawania uprawnień należy rozpatrzyć jeszcze jeden mechanizm. Otóż często zdarza się, że użytkownik chce aby jego klasa miała tymczasowo dane uprawnienia w imieniu klasy, która takich uprawnień nie ma. Przykładowo chcemy aby możliwość tworzenia gniazd (Socket) dana była tylko klasom pochodzącym z węzła sieci komputerowej, a nie bezpośrednio klasom pochodzącym z poszczególnych działów firmy. Uprzywilejowanie kodu polega na stworzeniu obiektu interfejsu PrivilegedAction lub PrivilegedExceptionAction i poinformowaniu kontrolera dostępu o tym fakcie poprzez wywołanie metody statycznej AccessController.doPvivileged() z argumentem będącym stworzonym właśnie obiektem. Kod uprzywilejowany umieszcza się w ciele jedynej metody interfejsów PrivilegedAction i PrivilegedExceptionAction, a mianowicie metody run(). Przykładowa konstrukcja wygląda następująco:

```
jakasmetoda() {
        ...
        AccessController.doPrivileged(new PrivilegedAction() {
           public Object run() {
              // uprzywilejowany kod np.:
              System.loadLibrary("awt");
              return null; // nic nie zwraca, ale zawsze musi występować
           }
        });
        ...
}
```

Oznaczając dany kod jako uprzywilejowany (privileged) nadaje się mu tymczasowo dostęp do zasobów, do których ma ustawione uprawnienia (permission), a do których nie ma uprawnień kod, który go wywołuje. Pytanie, które się natychmiast pojawia jest po co uprzywilejować kod, który ma uprawnienia? Otóż jest to istotne w kontekście kodu nieuprzywilejowanego: ładowanie kodu odbywa się do pamięci; ostatnio przywołany kod znajduje się najwyżej; kontroler dostępu wywołany jawnie lub nie, sprawdza uprawnienia (checkPermissions) poszczególnych kodów; jeżeli choć jeden fragment kodu nie ma określonych uprawnień wedle zasad bezpieczeństwa zwracany jest wyjątek AccessControlException, chyba że dany fragment jest zaznaczony jako uprzywilejowany; jeżeli tak jest to kontroler dostępu sprawdza czy zaznaczony kod ma uprawnienia do wykonania danej operacji i jeśli jest to prawdą to zakończony jest proces sprawdzania dostępu - nie sprawdza się dalszego kodu. Podsumowując: Jeżeli kod A o uprawnieniach ustawionych w plikach zasad bezpieczeństwa wywołuje kod B, którego uprawnienia są większe niż kodu A (tzn. kod A nie ma ustawionych takich uprawnień w plikach jak kod B), to wówczas przy ładowaniu kodu (najpierw ładowany jest A potem B, najpierw sprawdzany jest B potem A) nastąpi wyjątek, chyba że oznaczymy B jako kod uprzywilejowany, czyli kod B nie będzie rzutował na A. Jeżeli z każdym kodem jest związana dziedzina

uprawnień i jest tych dziedzin (kodów) m to proces sprawdzania kodu przebiega następująco:

```
i = m;
while (i > 0) {
   if (dziedzina kodu i nie posiada danych uprawnień)
      throw AccessControlException
   else if (kod zaznaczony jako uprzywilejowany)
      return; // koniec sprawdzania
   i = i - 1;
};
```

Zanim rozpatrzone zostaną zagadnienia kryptografii w Javie warto przedstawić kilka przykładów do powyższych rozważań. Powiedziane zostało, że w modelu bezpieczeństwa Java 2, każdy kod (dla aplikacji - o ile działa system zarządzania bezpieczeństwem) podlega kontroli wedle plików zasad bezpieczeństwa. Standardowe ustawienie bezpieczeństwa w pliku java.policy, które przedstawiono powyżej, umożliwia uruchomienie gniazda na portach wyższych niż 1024 (na portach użytkownika):

```
// allows anyone to listen on un-privileged ports
permission java.net.SocketPermission "localhost:1024-", "listen";
```

Ustawienie to uniemożliwia stworzenie serwera na portach mniejszych niż 1024. Oznacza to, że prezentowana wcześniej aplikacja SerwerEcho (w rozdziale poświęconym pracy w sieci) nie może działać przy wywołaniu jej z uruchomionym systemem zarządzania bezpieczeństwa (włączenie sprawdzania uprawnień). W celu sprawdzenia działania uprawnień należy wywołać:

```
java -Djava.security.manager SerwerEcho
```

Zwrócony zostanie wyjątek AccessControlException z podaniem typu uprawnień, których brak.
Warto pamiętać, że nie o wszystkich wyjątkach użytkownik programu może wiedzieć, ponieważ wyjątki mogą być obsługiwane w programie bez zwracania komunikatu.

W celu zapoznania się z mechanizmem ustawiania uprawnień za pomocą narzędzia policytool przedstawiony zostanie następujący przykład appletu:

Przykład 10.1:

```
//Zapisz.java:

import java.awt.*;
import java.io.*;
import java.lang.*;
import java.applet.*;

public class Zapisz extends Applet {
   String plik = "test";
```

```
   File f = new File(plik);
   DataOutputStream strumien_wy;

  public void init() {

   String osname = System.getProperty("os.name");
  }

  public void paint(Graphics g) {
       try {
         strumien_wy = new DataOutputStream(new BufferedOutputStream(new FileOutputStream(plik),128));
         strumien_wy.writeChars("Niespodzianka: Applet zapisal ten plik !!!\n");
         strumien_wy.flush();
         g.drawString("Zapisano plik " + plik + "; sprawdz!!! ", 10, 10);
       }
       catch (SecurityException e) {
         g.drawString("Applet Zapisz spowodowal blad bezpieczenstwa: " + e, 10, 10);
       }
       catch (IOException ioe) {
             g.drawString("Applet Zapisz spowodowal blad we/wy", 10, 10);
       }
   }
}// koniec public
```

Zapisz.html:

```
<HTML>
<HEAD>
   <TITLE>test Zapisz</TITLE>
</HEAD>
<BODY>
To jest przykładowy Applet:
<CENTER><APPLET CODE="Zapisz.class" WIDTH=600 HEIGHT=200
ALIGN=CENTER></APPLET></CENTER>
</BODY>
</HTML>
```

Applet ten tworzy plik tekstowy o nazwie "test" z zapisanym przykładowym tekstem:"Niespodzianka: Applet zapisal ten plik !!!". W przypadku domyślnych ustawień zasad bezpieczeństwa applet ten nie zapisze jednak pliku lecz zwróci wyjątek SecurityException. Można to sprawdzić wywołując skompilowany kod appletu przez:

appletviewer Zapisz.html

W celu ustawienia konkretnych uprawnień dla danego kodu należy wywołać narzędzie policytool poprzez wpisanie:

policytool

Pojawi się okno aplikacji Javy z dwoma polami Policy File oraz Keystore (które są puste jeśli nie ma ustawień użytkownika) oraz trzema przyciskami umożliwiającymi edycję plików uprawnień. W celu stworzenia nowego pliku uprawnień i nadania

uprawnień dla zapisu pliku o nazwie "test" należy wcisnąć klawisz Add Policy Entry, co spowoduje otwarcie nowego okna dialogowego, w którym można wpisać uprawnienia dla danego kodu. W nowym oknie dialogowym najpierw należy wybrać źródło kodu, któremu nadajemy uprawnienia (Code Base). Zostawiając to pole puste ustawimy uprawnienia dla dowolnego kodu.
Następne pole SignedBy umożliwia ustawienie identyfikacji jednostki autoryzującej dany kod. Ponownie zostawiając to pole puste ustawimy każdy kod jako autoryzowany. Teraz można ustalić typy dostępu. W tym celu wybrać należy przycisk AddPermission co spowoduje otwarcie trzeciego okna dialogowego z możliwością wyboru typu uprawnień z listy: Permission; oznaczenia celu działania danego uprawnienia (np. nazwa pliku) w polu Target Name, oraz wskazania typu akcji właściwej dla danego uprawnienia w polu Actions. Dla potrzeb tego przykładu należy wybrać z listy Permission:FilePermission, wpisać obok TargetName nazwę pliku czyli "test" oraz typ akcji "write". Po potwierdzeniu utworzenia nowych uprawnień (OK w oknie Permissions; Done w oknie Policy Entry) należy zapisać uprawnienia jako nowy plik np. moje_policy, poprzez wybór polecenia SaveAs z Menu okna Policy Tool. Po nagraniu pliku zasad bezpieczeństwa pojawi się w polu Policy File okna PolicyTool nazwa nowo stworzonego pliku bezpieczeństwa wraz ze ścieżką dostępu (URL). Po wykonaniu tych czynności należy zakończyć pracę z aplikacją PolicyTool i ponownie wywołać applet Zapisz.java poprzez:

```
appletviewer -J-Djava.security.policy=moje_policy Zapisz.html
```

Opcja -J-Djava.security.policy= umożliwia wywołanie danego appletu z tymczasowo przyjętym plikiem zasad bezpieczeństwa. Oczywiście uprawnienia w pliku moje_policy można przenieść do domyślnego pliku w swoim katalogu domowym lub można utworzyć w pliku java.security następne odwołanie do pliku zasad bezpieczeństwa poprzez wpisanie policy.url.3=file:/C:/sciezkadostepu/moje_policy. Wówczas nie trzeba podawać opcji dodatkowego pliku bezpieczeństwa, gdyż będzie on sprawdzany automatycznie. Uwaga: W ustawieniach lokalizacji plików istotna jest różnica zapisu ścieżki dostępu do danych zasobów w systemie plików właściwych dla danego systemu operacyjnego.

Uruchomienie appletu Zapisz.java wraz z dodatkowym plikiem zasad bezpieczeństwa spowoduje nagranie pliku "test"do lokalnego systemu plików.

10.4 Kryptografia

W celu omówienia możliwości zastosowania kryptografii w środowisku języka JAVA należy najpierw omówić podstawowe zagadnienia związane z kryptografią.
Do podstawowych zadań kryptografii należy zaliczyć:
- zapewnienie autentyczności autora obiektu (np. danych) - autoryzacja;
- zapewnienie autentyczności obiektu (np. danych);
- kodowanie danych.

W celu zapewnienia autentyczności autora obiektu np.danych czy kodu, należy stworzyć takie mechanizmy, aby można było stworzyć unikalny identyfikator wskazujący autora obiektu. Najczęstszym rozwiązanie tego problemu jest stworzenie dla autora unikalnego klucza, którym będzie on "zamykał" wszystkie generowane

przez siebie obiekty. Popularne klucze służące kodowaniu wykorzystywano już od dawna. Przykładowo książka oraz film "Klucz do Rebeki" o działaniach wywiadowczych w czasach II Wojny Światowej przedstawia wykorzystanie książki jako klucza do szyfrowania wiadomości. Obie zainteresowane strony: wywiadowca i centrala, posiadają tą samą książkę, która stanowi tłumaczenie odnośników przesyłanych pomiędzy stronami. Oznacza to, że zarówno nadawca jak i odbiorca posiadają dobrze im znany klucz. Jeżeli taki klucz (np. taka książka) jest unikalny i występuje tylko u odbiorcy i nadawcy, to wówczas odbiorca wie kto nadał daną wiadomość. Ta forma klucza nazywa się kluczem sekretnym, co oznacza, że klucz musi być ukryty zarówno przez nadawcę jak i odbiorcę. O wiele bardziej popularne obecnie jest wykorzystanie dwóch różnych kluczy dla potrzeb kryptografii: klucza prywatnego (sekretnego) oraz publicznego.
Idea jest prosta. Autor generuje parę ściśle ze sobą związanych kluczy, a następnie rozsyła wszystkim tym, którzy z nim współpracują klucze publiczne. Klucz prywatny autor przechowuje tak starannie jak klucz sekretny. Autor nadając wiadomość tworzy jej skrót (message digest - skrót wiadomości) korzystając z konkretnego algorytmu (tworzony jest zestaw cyfr, stąd często funkcje generujące skrót nazywa się funkcjami haszującymi - # - to oznaczenie cyfry). Następnie kod jest kodowany za pomocą klucza prywatnego autora. W ten sposób powstaje cyfrowy podpis wiadomości. Sprawdzenie cyfrowego podpisu wiadomości przez odbiorcę odbywa się poprzez ponowne stworzenie skrótu wiadomości a następnie zakodowaniu tego skrótu za pomocą klucza publicznego nadawcy. Jeżeli podpis wygenerowany przez nadawcę i podpis wygenerowany przez odbiorcę są zgodne to oznacza, że istotnie wiadomość nadał znany odbiorcy nadawca. Oprócz identyfikacji nadawcy proces ten zapewnia jeszcze jedną korzyść- zapewnienie autentyczności wiadomości. Otóż wygenerowany skrót wiadomości jest unikalny (powinien być), co oznacza, że dowolna zmiana wiadomości spowoduje zmianę skrótu, a co za tym idzie wystąpi niezgodność podpisów, czyli brak potwierdzenia autentyczności autora i danych. W celu zwiększenia efektywności zarządzania kluczami (gdzie trzymać klucze? czyje to są klucze?...) wprowadza się dwa podstawowe elementy: bazy kluczy oraz certyfikaty. Najlepsze rozwiązanie to bazy certyfikatów czyli zbiorów przechowujących: klucz publiczny nadawcy, oznaczenie nadawcy (imię, nazwa, ...), podpis cyfrowy wydawcy certyfikatu (np. odbiorca generuje podpis cyfrowy danych (klucz publiczny nadawcy) na podstawie swojego klucza prywatnego) oraz oznaczenie wydawcy certyfikatu (imię, nazwa). Zbiory certyfikatów są przechowywane i wymieniane w zależności od potrzeb. Sprawdzenie wiarygodności certyfikatu odbywa się poprzez sprawdzenie podpisu cyfrowego w nim zawartego lub często tworzy się "odcisk" certyfikatu czyli skrót certyfikatu i porównuje się skróty posiadanego certyfikatu i certyfikatu od wystawcy.
Klucze mają oczywiście jeszcze jedno podstawowe wykorzystanie - kodowanie danych. Otóż do tej pory omówione zostały mechanizmy zapewnienia autentyczności autora i obiektu - dane były nie kodowane. Dane można bezpośrednio kodować kluczem sekretnym (prywatnym), natomiast odbiorca może dekodować dane za pomocą klucza publicznego. Rozwiązanie to nie jest jednak zbyt dobre ponieważ wówczas każdy kto posiada klucz publiczny (a ponieważ jest on publiczny nie ma ograniczeń w dostępie) może odkodować wiadomość, czyli właściwie kodowanie danych nie przynosi spodziewanych rezultatów. Do kodowania danych wykorzystuje się jednak mechanizm odwrotny. Mając klucz publiczny odbiorcy kodujemy dane. Tak zakodowane dane odkodować może tylko właściciel klucza prywatnego odpowiadającego kluczowi publicznemu użytemu do kodowania wiadomości.

Do tej pory wskazane zostały pewne mechanizmy kryptografii, bez wymieniania metod matematycznych służących do kodowania, generacji skrótów czy kluczy. Praktycznie w kryptografii operuje się algorytmami, czyli konkretnie zrealizowanymi metodami. Algorytmy dostarczane są w odpowiednich pakietach (jar, zip) dostarczanych przez właściwego dostawcę (Provider). Korzystając z odpowiedniego algorytmu należy wskazać z jakiego pakietu (jeśli nie jest to pakiet domyślny) powinien być zastosowany dany algorytm.

10.4.1 Kryptografia w Javie

Począwszy od JDK 1.1 pojawiły się w javie mechanizmy kryptografii. W Javie 2 rozszerzono te mechanizmy. Wszystkie mechanizmy podzielono funkcjonalnie na dwie architektury: JCA - Java Cryptography Architecture oraz JCE - Java Cryptography Extension. JCA stanowi integralną część Java 2 API i umożliwia podstawowe operacje celem zapewnienia autentyczności autora i danych. Obsługuje więc następujące mechanizmy: skróty wiadomości (MessageDigest), podpisy cyfrowe (Signatures), certyfikaty (Certificates), generacja kluczy (KeyPairGenerator), przechowywanie kluczy (KeyStore). JCE natomiast jest rozszerzeniem Java 2 API i głównie służy do kodowania danych (a także m.in. do generacji i obsługi kluczy sekretnych). Dostęp do JCE niestety jest ograniczony ze względu na prawo eksportowe USA. Korzystać z tej biblioteki mogą jedynie obywatele USA i Kanady. Praktycznie rzecz ujmując warto omówić tylko pierwszy zestaw mechanizmów czyli JCA.
Jak wspomniano wcześniej wszystkie mechanizmy kryptografii bazują na konkretnych algorytmach wykorzystywanych do obliczeń (kodowania). Zestaw algorytmów i narzędzi jest pojmowany w Javie jako pakiet dostawcy (Cryptographic Service Provider package lub w skrócie provider package). Pakiet dostawcy wykorzystywany w danym środowisku Javy musi być zarejestrowany w odpowiednim pliku konfiguracyjnym tj. w pliku java.security, według składni:

security.provider.n=masterClassName

Standardowym pakietem wykorzystywanym dla potrzeb JCA jest pakiet "SUN" dostarczany wraz z dystrybucją JDK. Domyślnie więc w pliku java.security wystąpi zapis:

security.provider.1=sun.security.provider.Sun

W pakiecie "SUN" znajdują się implementacje różnych metod kryptografii: DSA (według NIST FIPS 186), MD5 i SHA-1 dla skrótów wiadomości, generator kluczy DSA, generator pseudo-losowy SHA1PRNG (według IEEE P1363), certyfikaty według X.509, JKS (nazwa implementacji zarządzania kluczami - keystore). Oczywiście można stosować dodatkowe pakiety dostawcy algorytmów kryptografii, przy czym przy inicjowaniu obiektu danej klasy mechanizmu kryptografii poprzez statyczną metodę getInstance() należy dodatkowo podać nazwę pakietu dostawcy. Przykładowo tworząc obiekt klasy MessageDigest można posłużyć się metodą public static MessageDigest getInstance(String algorithm, String provider) throws NoSuchAlgorithmException, NoSuchProviderException. Nazwę algorytmu oraz

nazwę dostawcy podaje się jako łańcuch znaków. Istotne jest właściwe rozróżnienie wielkości znaków gdyż metoda jest czuła na wielkość liter i w przypadku błędu zostanie zwrócony wyjątek. Praca z mechanizmami kryptografii w Javie zaczyna się więc prawie zawsze od inicjowania obiektu danego mechanizmu wywołując statyczną metodę getInstance z podaniem nazwy algorytmu i (jeśli to konieczne) nazwy pakietu. Sposób taki na realizację algorytmów kryptografii w programowaniu jest niezwykle uniwersalny. Pozwala bowiem na uniezależnienie się w znaczny sposób w kodzie programu Javy od tworzonych i rozwijanych algorytmów kryptografii. Dalsze przykłady w tym przewodniku będą opierały się na pakiecie dostawcy kryptografii "SUN".

10.4.2 Skróty wiadomości

Pierwszym mechanizmem kryptografii w Javie, jaki tu zostanie przedstawiony, będzie tworzenie skrótów wiadomości. Do tego celu stworzono klasę MessageDigest wraz z odpowiednimi jej metodami. W celu zainicjowania obiektu tej klasy (podobnie jak w innych przypadkach) wykorzystuje się statyczną metodę getInstance(), np.:

```
MessageDigest md = MessageDigest.getInstance("SHA")
MessageDigest md = MessageDigest.getInstance("MD5").
```

Następnym krokiem po inicjalizacji jest dostarczenie obiektowi danych do stworzenia skrótu. Odbywa się to poprzez podanie tablicy bajtów jako argumentu do jednej z metod update(), np. jeżeli dane tworzą tablicę bajtów t1, wówczas:

```
md.update(t1);
```

jeśli dane tworzą dwie tablice t1 i t2 to:

```
md.update(t1);
md.update(t2);
```

i tak dalej.
Obliczenie skrótu jest ostatnim krokiem i odbywa się na skutek wywołania metody digest(), np.

```
byte  skrot[];
skrot = md.digest();
```

Jeśli jest jedna tablica danych wówczas można skomasować dwa ostatnie kroki i wywołać inną metodę digest() z argumentem będącym tablicą danych np.:

```
skrot = md.digest(t1);
```

Jak pokazano na przykładach metoda digest() zwraca tablicę bajtów będącą właściwym skrótem wiadomości.

Poniżej przedstawiono przykładowy program umożliwiający stworzenie skrótu, wyświetlenie go oraz zapis wraz z wiadomoscią do pliku.

Przykład 10.2:

```
//ZapiszMD.java:

import java.io.*;
import java.security.*;

public class ZapiszMD {

 public static void main(String args[]){
  try{
   FileOutputStream strumien_wy=new FileOutputStream("testMD");
   MessageDigest md = MessageDigest.getInstance("SHA");
   ObjectOutputStream obiekt_wy = new ObjectOutputStream(strumien_wy);
   String wiadomosc = "Czy to sen? Czy to sun? Czy to Java?";
   byte buf[] = wiadomosc.getBytes();
   md.update(buf);
   byte skrot[] = md.digest();
   System.out.println("Oto skrot:"+skrot);
   obiekt_wy.writeObject(wiadomosc);
   obiekt_wy.writeObject(skrot);
  } catch(Exception e){
   System.out.println(e);
  }

 }
}// koniec public class ZapiszMD
```

Program ten zapisze w formie obiektów dwa elementy: dane oraz skrót. Powstały plik zostanie wykorzystany w poniższym przykładzie do weryfikacji danych na podstawie skrótu:

Przykład 10.3:

```
//CzytajMd.java:

import java.io.*;
import java.security.*;

public class CzytajMD {

 public static void main(String args[]){
  try{
      FileInputStream strumien_we=new FileInputStream("testMD");
      ObjectInputStream obiekt_we = new ObjectInputStream(strumien_we);
      Object ob = obiekt_we.readObject();
      if(!(ob instanceof String)) {
          System.out.println("Niewlasciwe dane w pliku");
          System.exit(-1);
      }
      String wiadomosc_we = (String) ob;
      String wiadomosc = "Czy to sen? Czy to sun? Czy to Java?";
      System.out.println("Oto wiadomosc otrzymana: "+wiadomosc_we);
      System.out.println("Oto wiadomosc oryginalna: "+wiadomosc);
```

```
        ob = obiekt_we.readObject();
        if(!(ob instanceof byte[])) {
            System.out.println("Niewlasciwe dane w pliku");
            System.exit(-1);
        }
        byte skrot_we[] = (byte []) ob;
        MessageDigest md = MessageDigest.getInstance("SHA");
        byte buf[] = wiadomosc_we.getBytes();
        md.update(buf);
        byte skrot[] = md.digest();
        System.out.println("Oto skrot otrzymany:"+skrot_we);
        System.out.println("Oto skrot wg danych otrzymanych:"+skrot);
        if (MessageDigest.isEqual(skrot, skrot_we))
            System.out.println("Wiadomosc niezmieniona");
        else
            System.out.println("Wiadomosc zmieniona");

    } catch(Exception e){
        System.out.println(e);
  }
 }
}// koniec public class CzytajMD
```

Program czytający z pliku dwa obiekty a następnie formatujący je do obiektów typu String (dla wiadomości) oraz byte[] (dla skrótu) umożliwia sprawdzenie czy przesłany skrót odpowiada przesłanej wiadomości. Dla celów edukacyjnych przedstawiony wyżej program zna oryginalną wiadomość stąd prezentacja na ekranie wiadomości oryginalnej i przesłanej upewnia w poprawnym użyciu skrótu. Warto przećwiczyć powyższe programy zmieniając nieznacznie (np. 1 literę) wiadomość, nie zmieniając skrótu (np. poprzez nagranie innego obiektu typu String niż wiadomość np. "Czy to san? Czy to sun? Czy to Java?" do pliku wraz z oryginalnym skrótem dla właściwej wiadomości, lub poprzez edycję pliku binarnego np. program "Dos Navigator" ->Edit i zmianę odpowiedniej wartości np. 65 na 61 czyli e w a). Mając plik ze zmienioną wiadomością i oryginalnym skrótem można wywołać ponownie program CzytajMD. W rezultacie otrzyma się informację, że wiadomość została zmieniona.

10.4.3 Kod autentyczności wiadomości - MAC

Powyższa, podstawowa implementacji klasy MessageDigest nie umożliwia jednak dobrego zabezpieczania autentyczności wiadomości. Związane jest to z tym, że ewentualny hacker może zmienić zarówno wiadomość jak i skrót. Oznacza to, że może on wprowadzić swoją wiadomość i skrót dla niej stworzony. Wówczas odbiorca nie znając wiadomości przekonany będzie, że zgodnie z poprawnością skrótu wiadomość jest oryginalna. W celu poprawy bezpieczeństwa autoryzacji danych wprowadza się tzw. kod autentyczności wiadomości (MAC - Message Authentication Code), będący przetworzonym skrótem. W Javie nie ma ściśle sprecyzowanych mechanizmów generacji kodu MAC jest więc wiele możliwych rozwiązań. Jedno z rozwiązań to kodowanie za pomocą kluczy skrótu, inne to używanie przez dwie zainteresowane strony (nadawca i odbiorca) tajnej frazy szyfrującej skrót. Frazę stanowi łańcuch znaków, który miesza się z oryginalną wiadomością. Przykładowo w celu zmodyfikowania programu zapisującego ZapiszMD.java w celu generacji i

zapisu kodu MAC zamiast skrótu należy przykładowo wykonać następujące polecenia:

```
...
String fraza = "To jest moja fraza";
byte f[] = fraza.getBytes();
...
md.update(f);
md.update(buf);  //buf to tablica bajtów według wiadomości
byte skrot[] =md.digest();
md.update(f);
md.update(skrot);
byte MAC[] = md.digest();

...
obiekt_wy.writeObject(MAC);
```

Powyższe rozwinięcie programu ZapiszMD.java generuje skrót będący interpretacją wiadomości i frazy, następnie tak powstały skrót jest traktowany jako zbiór danych, na podstawie którego (ponownie wraz z frazą) generuje się nowy skrót stanowiący kod autentyczności MAC. Jeżeli odbiorca zna frazę może (odpowiednio przetwarzając program CzytajMD.java) sprawdzić autentyczność wiadomości. Ponieważ hacker nie zna frazy (nie powinien) nie może zmodyfikować wiadomości tak, aby odbiorca o tym nie wiedział.

10.4.4 Klucze i podpis cyfrowy

Jak wspomniano wcześniej rozwinięciem skrótu jest podpis cyfrowy. Podpis cyfrowy (Signature) jest niczym innym jak skrótem, kodowanym kluczem prywatnym. Istotna jest więc tutaj znajomość pracy z kluczami. W JCA stosowane są głównie pary kluczy prywatny-publiczny. Ponieważ jednak JCE umożliwia stosowanie kluczy sekretnych, w Javie core API wprowadzono elementy dla pojedynczych kluczy. I tak nadrzędnym interfejsem jest interfejs Key definiujący metody służące do oznaczenia danego klucza (np. typ algorytmu, format. kodowanie). Ponieważ Sun w swej dystrybucji Javy dostarcza pakiet kryptografii "SUN" z implementacją DSA do tworzenia kluczy wprowadzono interfejsy specyficzne dla kluczy DSA tj. DSAKey, oraz z niego wywodzą się dwa potomne interfejsy DSAPrivateKey i DSAPublicKey. Interfejsy te udostępniają metodę umożliwiającą określenie parametrów p, q i g wykorzystywanych do generacji kluczy w metodzie DSA. Klasą (final class - czyli właściwie jest to zwykły kontener danych) obsługującą klucze jest KeyPair. Klasa ta zawiera jedynie dwie metody umożliwiające uzyskanie klucza prywatnego i publicznego: getPrivate(), getPublic(). W celu wygenerowania kluczy należy oczywiście użyć odpowiedniego algorytmu kodowania oraz algorytmu generatora liczb losowych (a właściwie pseudo-losowych). Dlatego stosuje się obiekty klasy KeyPairGenerator służące do generacji kluczy według przyjętego algorytmu np. DSA. Przykładowy fragment kodu umożliwiający otrzymanie pary kluczy o długości 1024 może wyglądać następująco:

```
KeyPairGenerator gen = KeyPairGenerator.getInstance("DSA");
```

```
gen.initialize(1024);
KeyPair kp = gen.generateKeyPair();
```

Ponieważ obiekt kp klasy kontenera KeyPair umożliwia pobranie klucza prywatnego i publicznego łatwo jest więc uzyskać te klucze celem ich dalszej obróbki. Poniższy program oblicza klucze prywatny i publiczny według algorytmu DSA pakietu SUN, a następnie klucze te są wyświetlane na ekranie:

Przykład 10.4:

```
//Klucze.java:

import java.security.*;

public class Klucze {

 public static void main(String args[]){

  try{
   KeyPairGenerator gen = KeyPairGenerator.getInstance("DSA");
   gen.initialize(512);
   KeyPair kp = gen.generateKeyPair();
   System.out.println("Oto klucz prywatny:"+kp.getPrivate());
   System.out.println("Oto klucz publiczny:"+kp.getPublic());

  } catch(Exception e){
   System.out.println(e);
  }

 }
}// koniec public class Klucze
```

Łatwo zauważyć, że metody println() wyświetlą na ekranie kilka parametrów tj. p, q, g, x oraz y. Wartość x to właśnie klucz prywatny, wartość y to klucz publiczny, natomiast wartości p,q, g to parametry DSA obliczone wcześniej dla pakietu SUN dla ustawień rozmiaru klucza 512, 768 i 1024 (dla DSA możliwe są ustawienia rozmiaru klucza będącego wielokrotnością 64, dla RSA możliwe są ustawienia rozmiaru klucza będącego wielokrotnością 8; w obu przypadkach wartość musi być większa niż 512). Obliczone wcześniej parametry znacznie przyspieszają proces generacji kluczy. Inicjowanie generatora kluczy może dodatkowo opierać się o zestaw bajtów podanych przez użytkownika. Najczęściej użytkownik jest zmuszony do zapisania tekstu, a dopiero później generowany jest zestaw kluczy. Przykład tego typu generacji kluczy ukazuje program Klucze2.java.

Przykład 10.5:

```
//Klucze2.java:

import java.security.*;
import java.io.*;

public class Klucze2 {
```

```
public static void main(String args[]){
 try{
  KeyPairGenerator gen = KeyPairGenerator.getInstance("DSA");
  SecureRandom los = SecureRandom.getInstance("SHA1PRNG");
  System.out.println("Podaj przykladowy tekst:");
  String seed = (new BufferedReader(new InputStreamReader(System.in))).readLine();
  los.setSeed(seed.getBytes());
  gen.initialize(512, los);
  KeyPair kp = gen.generateKeyPair();
  System.out.println("Oto klucz prywatny:"+kp.getPrivate());
  System.out.println("Oto klucz publiczny:"+kp.getPublic());

 } catch(Exception e){
  System.out.println(e);
 }

}
}// koniec public class Klucze2
```

W powyższym przykładzie program czeka na wprowadzenie tekstu wejściowego, który jest później wykorzystany w wywołaniu generatora liczb losowych, a docelowo w generatorze kluczy. Klucze są następnie wyświetlone na ekranie. Najczęściej parę kluczy generuje się raz, i są one umieszczane w odpowiedniej bazie danych (keystore) dla potrzeb ich wielokrotnego użycia przy autoryzacji każdej wiadomości.

Jak powiedziano wcześniej proste użycie skrótu nie gwarantuje zabezpieczenia autentyczności wiadomości. Jedną z omówionych wcześniej metod było użycie dodatkowych, tajnych fraz. Innym rozwiązaniem w Javie jest zastosowanie kluczy do kodowania skrótów. Otrzymujemy w ten sposób podpis cyfrowy w Javie. Inicjowanie podpisu cyfrowego w Javie jest więc niczym innym jak wywołaniem skrótu z kodowaniem kluczem prywatnym. Wywołanie obiektu klasy Signature (podpis cyfrowy) odbywa się podobnie jak to omawiano wcześniej, poprzez zastosowanie statycznej metody getInstance() z podaniem typu algorytmu i ewentualnie dostawcy. Przykładowe wywołanie może mieć następujący charakter:

```
Signature podpis = Signature.getInstance("SHA1withDSA");
```

Sama nazwa algorytmu wskazuje na typ operacji: po pierwsze generacja skrótu z wykorzystaniem algorytmu SHA1 oraz wykorzystanie klucza DSA. Obiekt klasy Signature posiada zawsze określony stan. Początkowo jest to stan bez inicjowania - UNINITIALIZED, a następnie możliwy jest jeden z dwóch stanów:
podpisywanie - SIGN oraz weryfikacja VERIFY. W celu stworzenia podpisu należy wprowadzić obiekt podpisu w stan SIGN za pomocą metody initSign(), np.:

```
podpis.initSign(kp.getPrivate());
```

Metoda initSign() wymaga więc jako atrybutu klucza prywatnego koniecznego do generacji podpisu. Kolejne dwa kroki w celu stworzenia podpisu to dostarczenie wiadomości do obiektu podpisu a następnie generacja tego podpisu:

```
podpis.update(wiadomosc);
byte kod_podpisu[] =podpis.sign();
```

W ten sposób wygenerowany został ciąg bajtów określający kod podpisu cyfrowego. W celu zobrazowania całego procesu można posłużyć się następującym przykładem:

Przykład 10.6:

```
//Podpis.java:

import java.security.*;
import java.io.*;

public class Podpis {

 public static void main(String args[]){
  try{
   String tekst = "Czy to sen? Czy to sun? Czy to Java?";
   byte wiadomosc[] = tekst.getBytes();
   MessageDigest md = MessageDigest.getInstance("SHA");
   md.update(wiadomosc);
   byte skrot[] = md.digest();
   KeyPairGenerator gen = KeyPairGenerator.getInstance("DSA");
   SecureRandom los = SecureRandom.getInstance("SHA1PRNG");
   System.out.println("Podaj przykladowy tekst:");
   String seed = (new BufferedReader(new InputStreamReader(System.in))).readLine();
   los.setSeed(seed.getBytes());
   gen.initialize(1024, los);
   KeyPair kp = gen.generateKeyPair();
   System.out.println("Oto klucz prywatny:"+kp.getPrivate());
   System.out.println("Oto klucz publiczny:"+kp.getPublic());
   Signature podpis = Signature.getInstance("SHA1withDSA");
   podpis.initSign(kp.getPrivate());
   podpis.update(wiadomosc);
   byte kod_podpisu[] =podpis.sign();
   System.out.println("Oto podpis cyfrowy: "+kod_podpisu);
   System.out.println("A tak wyglada sam skrot: "+skrot);

  } catch(Exception e){
   System.out.println(e);
  }

 }
}// koniec public class Podpis
```

Powyższy przykład definiuje na początku tekst oraz wiadomość będącą ciągiem bajtów związanych z tekstem. W celu ukazania różnicy pomiędzy skrótem a podpisem, po definicji wiadomości generowany jest skrót. Następnie po generacji kluczy tworzony jest podpis cyfrowy dla danej wiadomości i jest on wyświetlany w porównaniu ze skrótem.
Odbiorca otrzymując wiadomość oraz podpis cyfrowy musi zweryfikować podpis, w celu sprawdzenia autentyczności wiadomości oraz określeniu autora tekstu. W tym celu obiekt podpisu należy wprowadzić w stan weryfikacji - VERIFY - wywołując metodę initVerify() z podaniem klucza publicznego autora, np.:

```
podpis.initVerify(kp.getPublic());
```

Następnie dostarcza się dane do obiektu i wywołuje się metodę weryfikacji verify():

```
podpis.update(wiadomosc);
boolean test = podpis.verify(kod_podpisu);
```

W przypadku poprawności podpisu metoda verify() zwraca true w przeciwnym przypadku false.

W większości przypadków, jak wspomniano wcześniej, klucze nie są generowane za każdym razem przy tworzeniu podpisu, lecz są przechowywane w specjalnej bazie danych. Dla Javy taką bazą danych może być keystore, przechowująca klucze i certyfikaty. W celu dostępu do bazy kluczy stosuje się mechanizmy dostarczane przez klasę KeyStore lub poprzez użycie programów narzędziowych dostarczanych z dystrybucją Javy tj., "keytool", "jarsigner" (podpisywanie archiwuw JAR) oraz "policytool". Baza danych "keystore" to nic innego jak plik o właściwej strukturze. W celu identyfikacji, "keystore" dla pakietu SUN posiada identyfikator JKS. Inicjowanie obiektu klasy KeyStore odbywa się podobnie jak dla pozostałych mechanizmów kryptografii w Javie poprzez przywołanie metody statycznej getInstance(). Następnie należy załadować dane aktualnie przechowywane w bazie do pamięci poprzez wykorzystanie metody load(). Metoda load() specyfikuje strumień wejściowy (skąd czytać) oraz ewentualnie hasło dostępu. Każda pozycja w bazie keystore jest identyfikowana przez unikalną nazwę (ang. alias). W celu uzyskania nazwy można posłużyć się metodą aliases(). W celu określenia czy dana pozycja w bazie jest kluczem czy certyfikatem należy wykorzystać metody typu boolean tj.: isKeyEntry() oraz isCertificateEntry() z podaniem jako argumentu odpowiedniej nazwy pozycji, o którą pytamy. Aby stworzyć nową pozycję w bazie keystore musimy najpierw związać nazwę z daną pozycją poprzez metody: setCertificateEntry() oraz setKeyEntry() podając jako argument odpowiednio certyfikat lub klucz, alias, oraz ewentualnie hasło. W razie potrzeby możemy usunąć pozycję z bazy wykorzystując metodę deleteEntry() z podaniem aliasu pozycji do usunięcia. Wszystkie te operacje są właściwe tylko wówczas, gdy baza keystore jest załadowana do pamięci. W celu nagrania nowo stworzonej lub zaktualizowanej bazy na dysku należy przywołać metodę store() podając odpowiedni strumień wyjścia i hasło. Oprócz przechowywania kluczy w bazie konieczna jest również możliwość pobierania z bazy kluczy celem ich wykorzystania np. dla generacji podpisów. W tym celu mając w pamięci załadowaną bazę należy przywołać metodą getKey() lub getCertificate() odpowiedni klucz lub certyfikat poprzez podanie właściwej nazwy jako argumentu metody.

Na zakończenie omawiania mechanizmów kryptografii w JCA warto zwrócić uwagę na istnienie specjalnych wyjątków poszczególnych mechanizmów. Przykładowo: DigestException, KeyException, SignatureException, itp. Wyjątki te najczęściej występują przy niezgodności inicjowania elementu wedle przyjętego algorytmu danego mechanizmu np. błąd w czasie wywoływania metody sign() (SignatureExcpetion) lub w czasie wykonywania metody digest() (DigestException), itd.

10.4.5 Kodowanie danych

Podsumowując zagadnienia kryptografii w Javie należy powiedzieć, że mechanizmy służące zapewnianiu autentyczności są dostarczane w sposób prosty, tak więc zastosowanie ich w programach nie wymaga specjalistycznej wiedzy programisty. Jakość kodowania jest zależna od zewnętrznych algorytmów, tak więc czysty kod
Javy jest niezwykle uniwersalny. Niestety brak możliwości kodowania danych (JCE - tylko w obrębie USA i Kanady) stanowi poważny problem. Kodowanie danych musi odbywać się więc zewnętrznie (inne aplikacje nie w Javie), a zakodowane dane mogą być następnie dostarczane do aplikacji Javy. Można również wykorzystać opracowane przez inne firmy pakiety do kodowania danych, np.:

Australian Business Access
ABA JCE
http://www.aba.net.au/solutions/crypto/jce.html

Baltimore Technologies
J/CRYPTO
http://www.baltimore.ie/products/jcrypto/index.html

Cryptix
Cryptix
http://www.cryptix.org/products/cryptix31/index.html

DSTC
JCSI
http://security.dstc.edu.au/projects/java/release3.html

Entrust(R) Technologies
Entrust/Toolkit Java Edition
http://www.entrust.com/toolkit/java/index.htm

Forge Research
Forge Security Provider
http://www.forge.com.au/products/crypto/index.html

IAIK
IAIK-JCE
http://jcewww.iaik.tu-graz.ac.at/jce/jce.htm

RSA Data Security, Inc.
Crypto-J
http:// www.rsasecurity.com/products/bsafe/cryptoj.html

Należy jednak zawsze zwracać uwagę na warunki licencjonowania, prawa eksportu kodu, oraz dostarczaną funkcjonalność kodu.