# 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
```

---

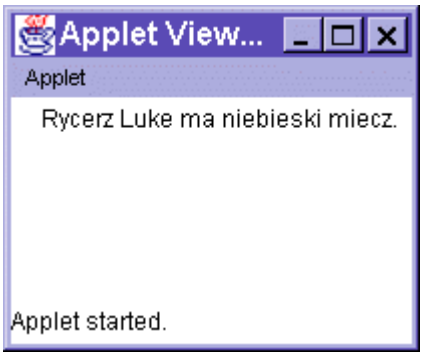
Jedi2.html :

```
<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.