Rozdział 3 Programowanie obiektowe

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 IR=Rycerz.liczbaRycerzy;
        System.out.println(„Liczba Rycerzy Jedi w Republice wynosi: „+IR);
        return IR;
}
```

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 składa 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.