Spis treści

Temat: Dziedziczenie ...
Celem wykładu jest pokazanie podstawowych mechanizmów stosowanych w programowaniu zorientowanym obiektowo. Głównie chodzi o dziedziczenie, rozszerzanie klas, przesłanianie i przeciążanie metod.

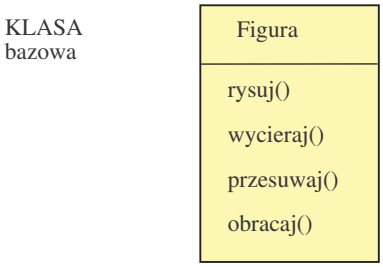
Podstawa: Bruce Eckel, Thinking in Java, Second Ed., Prentice Hall, 1998
The JavaLanguage Environment, A white Paper, Sun, Oct. 1995;

1 Dziedziczenie

Zadeklarujemy klasę `Figura` oraz cztery metody, które w zamyśle, realizują rysowanie, wycieranie, obrót i przesuwanie figur geometrycznych.

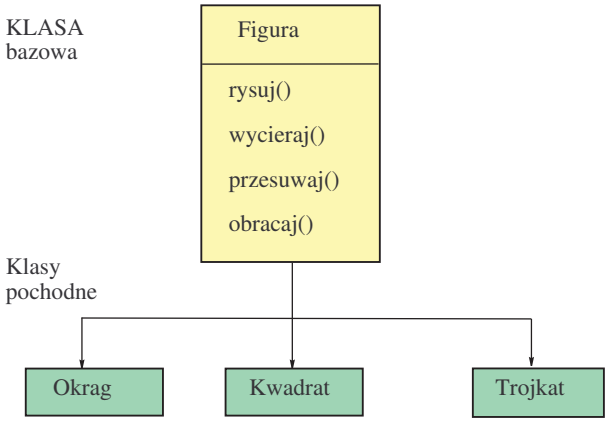
```
class Figura extends Object {
   void rysuj() {
     //// rysowanie
   }
   void wycieraj() {
     //// ...
   }
   void obracaj() {
     //// ...
   }
   void przesuwaj() {

   }

}
```

Klasę Figura możemy przedstawić przy pomocy nazwanego prostokąta. Jego dolna część (pod kreską) zawiera nazwy metod klasy. Na rysunku widzimy przykład klasy `Figura`.



Klasa bazowa

Od klasy bazowej `Figura` można wywieść inne bardziej szczególne klasy takie jak np, okręgi, kwadraty i trójkąty.



Klasa bazowa i klasy potomne

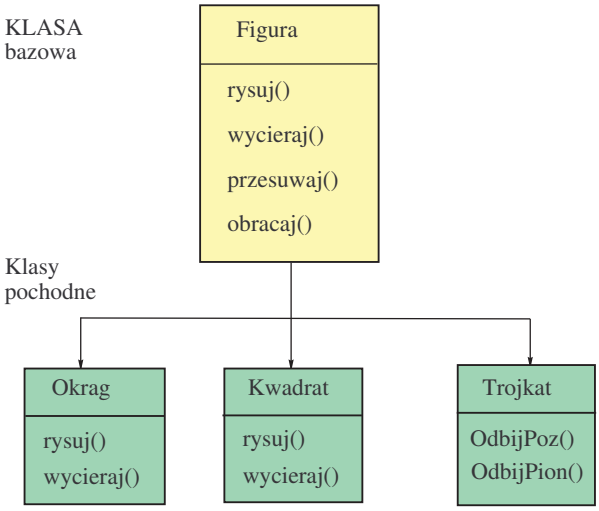
```
class Kwadrat extends Figura {
  //// tutaj jakies deklaracje
}
```

Te nowe klasy dziedziczą wszystkie pola i metody klasy przodka `Figura`. Oznacza to, że metody `rysuj()` itd. są w nich dostępne tak samo jak w biektach typu `Figura`.

2 Wymiana metod i poszerzanie klas

Metody odziedziczone przez klasy potomne można wymienić na nowe (*ang.* overriding) lub można dodawać w nowych klasach całkiem nowe metody.

Klasy potomne otrzymane metoda pozszerzania (`extends`) mają dostęp do wszystkich metod klasy bazowej. Jeśli o klasie pomyślimy jako o typie danych to proces ten jest poszerzaniem typu danych. Zarówno okręgi (`Okrag`) jak też kwadraty (`Kwadrat`) i trójkąty (`Trojkat`) są figurami (`Figura`), ale mają dodatkowe, własne cechy. W przypadku trójkątów może to byc np. możliwość ich poziomego lub pionowego odbijania (np. wzgledem prostej pionowej lub poziomej przechodzącej przez środek trójkąta). Zrealizujemy te nowe możliwości wprowadzając nowe procedury tak jak to pokazano na diagramie.



Klasa bazowa i klasy potomne.

Klasa Trojkat została pozszerzona i posiada dwie nowe metody odbijPoz() oraz odbijPion(). Metody rysuj() i wycieraj() klas Okrag i Kwadrat zostały zdefiniowane od nowa (*ang.* overrided).

Można tego dokonać tak

```
class Trojkat extends Figura {

  void odbijPion() {
     //// polecenia
  }

  void odbijPoz() {
    //// polecenia
  }
}
```

Dodatkowo, zmieniliśmy juz istniejące procedury klasy `Figura`, a więc procedury `rysuj(), wycieraj()` itp. Tak wygląda to w nowej klasie `Kwadrat, ....` Kwadrat wycieramy i rysujemy w specjalny sposób (inaczej niż okrąg). Podobnie jest z okręgiem.

```
class Kwadrat extends Figura {

  void rysuj() {
     // polecenia
  }

  void wycieraj() {
    // polecenia
  }
}
```

Tego typu praktyka pozwala rozszerzać i korygować istniejące już klasy (sprawdzone) bez konieczności interwencji do ich kodów źródłowych. Nowe klasy maja w ten sposób wszystkie potrzebne cechy istniejące w przodkach (klasach bazowych) i nowe cechy, których przodkowie nie posiadają, a więc charakterystyczne tylko dla wybranych potomków.

Deklaracja obiektów nowych klas przebiega tak samo jak poprzednio. Na przykład, deklarowanie obiektów `Kwadrat` odbywa sie wg. schematu

```
Kwadrat k1 = new Kwadrat();
Kwadrat k2 = new Kwadrat();
...
//// itd
...

k1.rysuj();
k2.rysuj();
k1.obracaj();
...
```

Uwaga! Wymiana, przesłanianie metody następuje tylko wtedy gdy przy jej powtórnym deklarowaniu w klasie potomka nie zmienimy nagłówka metody, tzn. jeśli liczba parametrów formalnych metody, ich typy i typ samej metody nie zmienią się. W innym wypadku mamy doczynienia z przeciążaniem metod,

3 Przeciążanie metod

Jeśli nagłówek w nowej deklaracja metody, oprócz samej nazwy metody, zawiera zmiany w porównaniu z już istniejąca metodą przodka, to metoda nie jest przesłaniana (wymieniana) lecz istnieje jako inna metoda. Możemy mieć wiele metod o tej samej nazwie lecz różnej liczbie parametrów formalnych lub różnych typach parametrów formalnych. W takim przypadku mamy doczynienia ze zjawiskiem przeciążania metod. Najczęściej przeciążanymi metodami są konstruktory.

P r z y k ł a d. 1.

```
////
//// overloading
class Overloading {
  public static void main(String[] args){
    System.out.println(dodaj(1, 2));
    System.out.println(dodaj(1f, 2f));
  }
  static String dodaj(int x, int y) {
    System.out.println("int");
    return String.valueOf(x+y);   // konwersja do łańcucha
  }
```

```
  static String dodaj(float x, float y) {
    System.out.println("float");
    return String.valueOf(x+y);   // konwersja do łańcucha
  }
}
////

Wynikiem działania programu jest

int
3
float
3.0
```

Wynik programu pozwala sądzić, że działały obie metody `dodaj(...)`, każda w dpowiedniej sytuacji. Java^{TM} rozróżnia oba przypadki. Ten przykład świadczy o tym, że mamy "wieloznacznąńazwę `dodaj(...)`, a więc jest tutaj realizowane przeciążenie metody.

4 Jednokrotne dziedziczenie

W Java^{TM}, inaczej niż w `C++`, dziedziczenie jest jednokrotne, tzn. każda klasa może dziedziczyć tylko od jednego przodka. Zamiast wielokrotnego dziedziczenia wprowadzono tzw. interfejsy (*ang.* interface), sprzęgi, które pozwalają na zastąpienie pewnych cech wielokrotnego dziedziczenia. Interfejs nie jest definicją obiektu, lecz definicją zbioru metod, które można zastosowac w jednym lub wielu obiektach. Interfejs jest tylko deklaracją metod i stałych. Nie można w nim deklarować zmiennych.

5 Polimorfizm

Załóżmy, że zdefiniowaliśmy klasy `Trojkat, Okrag` i `Kwadrat` tak jak wyżej. Chcemy teraz wykonac pewne prace z figurami. Ciąg tych prac jest zawsze taki sam: rysujemy figurę i obracamy ją. Z góry nie wiemy jakich figur bedzie to dotyczyć. Czy można to zrobić w jednej metodzie? Nazwijmy ją `wykonajPrace()`.

```
void wykonajPrace(Figura f) {

  f.rysuj();
  f.obracaj();
  //// ...

}

...
Okrag   o = new Okrag();
Trojkat t = new Trojkat();
Kwadrat k = new Kwadrat();
```

```
...
wykonajPrace(t);
wykonajPrace(o);
wykonajPrace(k);
...
```

Wywołanie metody `wykonajPrace(...)` automatycznie robi to co potrzeba. Ponieważ `o, k, w` są typu `Figura` to metoda `wykonajPrace(...)` traktuje je wszystkie tak, jak należy. Opisany mechanizm nosi nazwę polimorfizmu. Zauważmy, że przed wywołaniem metoda `wykonajPrace()` nie wie z jakim obiektem będzie miała doczynienia. Nie wie czy będzie to trójkąt, okrąg czy coś jeszcze. Rozstrzyga się to w ostatnim momencie. Odpowiednie metody `rysuj()` i `obracaj()` sa dobierane w chwili gdy dany obiekt *znajdzie się wewnątrz metody*. Nosi to nazwę dynamicznego wiązania metod, późnego wiązania lub viązania w czasie wykonywania programu. Realizacja tego typu zachowań w przypadku Java^{TM} jest bardzo wygodna z punktu widzenia programisty, czytelności programu i możliwości jego rozbudowy.

6 Kontrola dostępu

Istnieją cztery kategorie określające prawa dostępu do metod lub zmiennych obiektu. Są to `public`, `protected`, `private` oraz kategoria *bez nazwy* określana jako *przyjazna*.

Jeśli zmienna nie jest zadeklarowana z użyciem jednego z trzech pierwszych słów to należy do kategorii zmiennych przyjaznych, Jest to zmienna, która jest dostępna we wszystkich obiektach danego pakietu (package). Organizacja klas w pakiety jest wygodna i zwiększa czytelność programu (patrz dalej).

W przypadku deklaracji `public`, zmienna (metoda) jest dostępna wszędzie.

Zmienna lub metoda kategorii `protected` jest dostępna tylko w podklasach danej klasy i nigdzie więcej.

W kategorii `private`, zmienne lub metody są dostępne tylko w klasie , w której zostały zadeklarowane i nigdzie więcej.

7 Zmienne klasowe i metody klasowe

Zwyczajne zmienne deklarowane w klasie są zmiennymi instancyjnymi - każda taka zmienna istnieje w każdym oddzielnym obiekcie utworzonym z tej klasy.

Zmienne klasowe są natomiast zmiennymi wspólnymi dla wszystkich obiektów tworzonych z tej klasy. Są one lokalne w klasie. Istnieje pojedyncza kopia kazdej zmiennej lokalnej.

Zmienne klasowe deklarujemy z użyciem słowa `static`.

```
class Prostokat extend Object {
  static final int version = 2;
  static final int revision = 0;
  //...
  ...
```

```
}
```

Słowo final oznacza, że zmienne klasowe w klasie `Prostokat` są stałymi.

Zupełnie podobnie rzecz się ma z metodami klasowymi. Wspólne metody, np. metoda wyznaczania wielkości okna aplikacji, są pojedynczymi metodami w klasie. Metody klasowe mogą operowac tylko na zmiennych klasowych i nie mają dostępu do zmiennych instancyjnych.

8 Metody i klasy abstrakcyjne

Rozważmy klasę abstrakcyjną. Jest to klasa, w której zdefiniowano metody (w zasadzie ich nagłówki), których nie zaimplementowano. Są one tylko zgłoszone i nie robią nic poza tym. W podklasach tej nadklasy metody te muszą byc przedefiniowane yak, by coś wykonywały.

Weźmy pod uwagę następujący przykład. Załóżmy, że piszemy program, rysujący figury geometryczne. W którymś momencie dochodzimy do wniosku, że wygodnie będzie zadeklarować klasę `Figura`, w której oprócz metod ustawiających kolory linii, kolory wypełnienia figur itd., przydatna będzie metoda `rysuj()`, która bedzie różna dla różnych figur, inna dla okręgu, a inna dla prostokąta. Definiujemy wiec *pustą* metodę `rysuj()`. Jej implementacja skonkretyzuje się w podklasach takich, jak `Okrąg`, `Prostokat` itd., a więc w przypadku kjonkretnych figur.

```
abstract class Figura extends Object {
  protected Punkt ld;
  protected Punkt pg;
  Color lk, ink;

  public void setPosition( Punkt ld, Punkt pg ) {
    this.ld = ld;
    this.pg = pg;
  }
  abstract void rysuj() {
  }
  public void setLineColor(Color k) {
    lk = k;
  }
  public void set setInsideColor(Color k) {
    ink = k;
  }
}
```

Ponieważ klasa `Figura` jest klasą `abstract` więc nie możemy utworzyć obiektów tej klasy. Można natomiast utworzyć klasy potomne (podklasy) i następni zadeklarować nowe metody `rysuj()` właściwe dla tych podklas. Napiszmy dla przykłady podklasę `Prostokąt`.

```
class Prostokat extends Figura {
  void rysuj() {
    moveTo(ld.x, ld.y);
    lineTo(ld.x, pg.y);
    lineTo(pg.x, pg.y);
    ...
  }
}
```

W ten sposób, klasa `Prostokat` posiada wszystkie metody klasy `Figura` i wie jak rysować prostokąt.

9 Podsumowanie

Najważniejszymi cechami programowania obiektowego jest **enkapsulacja** - ukrywanie informacji i modularność, **polimorfizm** - różne zachowanie sie obiektów w zależności od ich natury, **dziedziczenie** - przejmowanie metod i pól nadklasy w definiowanych podklasach, **dynamiczne wiązanie** - zapewnia maksymalną elastyczność w czasie wykonywania sie programu.