



Politechnika Wroclawska



Wielokrotne wykorzystanie klas

Martin Suszczyński 140893



Wstęp



<http://www.netbeans.org/>



<https://pwrjug.dev.java.net/>



Sun Academic Initiative

- Co oferuje SAI
 - ogromny zbiór Web kursów
 - brak opłat wpisowych oraz licencyjnych
 - certyfikaty Java, Solaris
 - practice exams
 - rejestracja online:

<http://learningconnection.sun.com>

Company Name: SAI-Wroclaw University of Technology

Company ID: CUS-0000096569



Wprowadzenie do tematu

- Jedna z najbardziej użytecznych cech Javy jest możliwość powtórnego wykorzystania kodu. Aby ta cecha zasługiwała na uwagę musi być czymś więcej niż zwykłym kopiowaniem kodu i dopasowywaniem go do potrzeb



Składnia kompozycji

- Klasa SprinklerSystem skomponowana z obiektu source, który jest zbudowany z klasy WaterSource
- toString() - metoda specjalna

```
1  //: reusing/SprinklerSystem.java
2  // Kompozycja w zżubie wielokrotnego wykorzystania kodu.
3
4  class WaterSource {
5      private String s;
6  WaterSource() {
7      System.out.println("WaterSource()");
8      s = "Skonstruowano";
9  }
10 public String toString() { return s; }
11 }
12
13 public class SprinklerSystem {
14     private String valve1, valve2, valve3, valve4;
15     private WaterSource source = new WaterSource();
16     private int i;
17     private float f;
18     public String toString() {
19         return
20             "valve1 = " + valve1 + " " +
21             "valve2 = " + valve2 + " " +
22             "valve3 = " + valve3 + " " +
23             "valve4 = " + valve4 + "\n" +
24             "i = " + i + " " + "f = " + f + " " +
25             "source = " + source;
26     }
27     public static void main(String[] args) {
28         SprinklerSystem sprinklers = new SprinklerSystem();
29         System.out.println(sprinklers);
30     }
31 } /* Output:
```

```

1  //: reusing/Bath.java
2  // Inicjalizacja wewnątrz konstruktora w przypadku kompozycji.
3  import static net.mindview.util.Print.*;
4
5  class Soap {
6      private String s;
7      Soap() {
8          print("Soap()");
9          s = "Skonstruowany";
10     }
11     public String toString() { return s; }
12 }
13
14 public class Bath {
15     private String // Inicjalizacja w miejscu definicji:
16         s1 = "Radosny",
17         s2 = "Radosny",
18         s3, s4;
19     private Soap castille;
20     private int i;
21     private float toy;
22     public Bath() {
23         print("Wewnątrz Bath()");
24         s3 = "Uradowany";
25         toy = 3.14f;
26         castille = new Soap();
27     }

```

```

28 // Blok inicjalizacji egzemplarza:
29 { i = 47; }
30 public String toString() {
31     if(s4 == null) // Inicjalizacja leniwa:
32         s4 = "Uradowany";
33     return
34         "s1 = " + s1 + "\n" +
35         "s2 = " + s2 + "\n" +
36         "s3 = " + s3 + "\n" +
37         "s4 = " + s4 + "\n" +
38         "i = " + i + "\n" +
39         "toy = " + toy + "\n" +
40         "castille = " + castille;
41 }
42 public static void main(String[] args) {
43     Bath b = new Bath();
44     print(b);
45 }
46 } /* Output:
47 Wewnątrz Bath()
48 Soap()
49 s1 = Radosny
50 s2 = Radosny
51 s3 = Uradowany
52 s4 = Uradowany
53 i = 47
54 toy = 3.14
55 castille = Skonstruowany
56 *///:~

```




Składnia dziedziczenia

```
1  //: reusing/Detergent.java
2  // Składnia i właściwości dziedziczenia.
3  import static net.mindview.util.Print.*;
4
5  class Cleanser {
6      private String s = "Cleanser";
7      public void append(String a) { s += a; }
8      public void dilute() { append(" dilute()"); }
9      public void apply() { append(" apply()"); }
10     public void scrub() { append(" scrub()"); }
11     public String toString() { return s; }
12     public static void main(String[] args) {
13         Cleanser x = new Cleanser();
14         x.dilute(); x.apply(); x.scrub();
15         print(x);
16     }
17 }
18
```

```
19  public class Detergent extends Cleanser {
20      // Zmiana metody:
21      public void scrub() {
22          append(" Detergent.scrub()");
23          super.scrub(); // Wywołanie wersji z klasy bazowej
24      }
25      // Uzupełnienie interfejsu o nowe metody:
26      public void foam() { append(" foam()"); }
27      // Test nowej klasy pochodnej:
28      public static void main(String[] args) {
29          Detergent x = new Detergent();
30          x.dilute();
31          x.apply();
32          x.scrub();
33          x.foam();
34          print(x);
35          print("Test klasy bazowej:");
36          Cleanser.main(args);
37      }
38      /* Output:
39      Cleanser dilute() apply() Detergent.scrub() scrub() foam()
40      Test klasy bazowej:
41      Cleanser dilute() apply() scrub()
42      *///:~
```



Inicjalizacja klasy bazowej

Dziedziczenie to nie tylko skopiowanie klasy bazowej

Kiedy tworzymy obiekt klasy pochodnej zawiera on w sobie klasę bazowa jako podobiekt

```
1  //: reusing/Cartoon.java
2  // Wywołania konstruktora przy dziedziczeniu.
3  import static net.mindview.util.Print.*;
4
5  class Art {
6  Art() { print("Konstruktor klasy Art"); }
7  }
8
9  class Drawing extends Art {
10 Drawing() { print("Konstruktor klasy Drawing"); }
11 }
12
13 public class Cartoon extends Drawing {
14 public Cartoon() { print("Konstruktor klasy Cartoon"); }
15 public static void main(String[] args) {
16     Cartoon x = new Cartoon();
17 }
18 } /* Output:
19 Konstruktor klasy Art
20 Konstruktor klasy Drawing
21 Konstruktor klasy Cartoon
22 *///:~
23
```




Konstruktor z argumentami

Jeśli klasa nie zawiera parametrów domyślnych lub gdy chcemy wywołać sparametryzowany konstruktor klasy bazowej, trzeba zastosować słowo `super` i podać odpowiednie argumenty

Kompilator zmusza do zamieszczenia wywołania konstruktora klasy bazowej jako pierwszej instrukcji w ciele konstruktora klasy pochodnej

```
1  //: reusing/Chess.java
2  // Dziedziczenie, konstruktory i argumenty.
3  import static net.mindview.util.Print.*;
4
5  class Game {
6  Game(int i) {
7      print("Konstruktor klasy Game");
8  }
9  }
10
11 class BoardGame extends Game {
12 BoardGame(int i) {
13     super(i);
14     print("Konstruktor klasy BoardGame");
15 }
16 }
17
18 public class Chess extends BoardGame {
19 Chess() {
20     super(11);
21     print("Konstruktor klasy Chess");
22 }
23 public static void main(String[] args) {
24     Chess x = new Chess();
25 }
26 } /* Output:
27 Konstruktor klasy Game
28 Konstruktor klasy BoardGame
29 Konstruktor klasy Chess
30 *///:~
31
```



Zapewnienie poprawnego sprzątnia

- Jeśli trzeba po czymś posprzątać, to czasami lepiej nie polegać na odsmiecaczu, tylko samemu powinno się dopisać odpowiednia metodę.
- Dodatkowo należy chronić się przed wyjątkami poprzez zamieszczenie takiego sprzątnia w klauzuli finally



Ukrywanie nazw

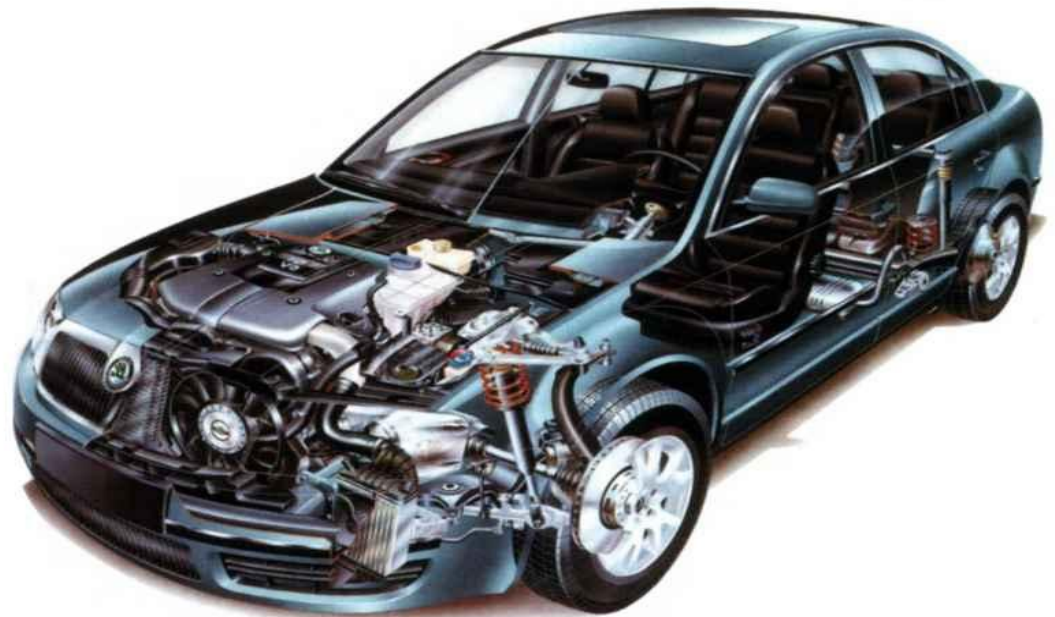
```
1  //: reusing/Hide.java
2  // Przeciążanie nazwy metody klasy bazowej w klasie
3  // pochodnej nie ukrywa wersji z klasy bazowej.
4  import static net.mindview.util.Print.*;
5
6  class Homer {
7  char doh(char c) {
8      print("doh(char)");
9      return 'd';
10 }
11 float doh(float f) {
12     print("doh(float)");
13     return 1.0f;
14 }
15 }
16
17 class Milhouse {}
18
19 class Bart extends Homer {
20 void doh(Milhouse m) {
21     print("doh(Milhouse)");
22 }
23 }
24
```

```
25 public class Hide {
26     public static void main(String[] args) {
27         Bart b = new Bart();
28         b.doh(1);
29         b.doh('x');
30         b.doh(1.0f);
31         b.doh(new Milhouse());
32     }
33 } /* Output:
34 doh(float)
35 doh(char)
36 doh(float)
37 doh(Milhouse)
38 *///:~
39
```



Wybor miedzy kompozycja a dziedziczeniem

- Oba mechanizmy pozwalają na zamieszczanie podobiektów wewnątrz nowo tworzonej klasy
- „jest”
- „ma”





Przyrostowe tworzenie oprogramowania

- Rozwój oprogramowania jest procesem przyrostowym, tj np. nauka
- Trzeba pamiętać, że dziedziczenie służy wyrażeniu związku oznaczającego, że „ta nowa klasa jest typu tamtej klasy”



Rzutowanie w górę

Konwersja referencji do obiektu klasy Wind na referencje do obiektu Instrument nazywamy „rzutowaniem w gore”

Jedynie, co może się zdarzyć z interfejsem klasy podczas rzutowania w gore, to utrata metod, ale nie jego rozszerzenia

Czy potrzebuje rzutowania w gore ??

```
1  //: reusing/Wind.java
2  // Dziedziczenie a rzutowanie w górę.
3
4  class Instrument {
5  public void play() {}
6  static void tune(Instrument i) {
7      // ...
8      i.play();
9  }
10 }
11
12 // Obiekty Wind są równocześnie typu Instrument
13 // bo mają identyczny interfejs:
14 public class Wind extends Instrument {
15 public static void main(String[] args) {
16     Wind flute = new Wind();
17     Instrument.tune(flute); // Rzutowanie w górę
18 }
19 } ///:~
20
```




Słowo kluczowe final

- Słowo kluczowe final dostępne w Javie może mieć różne znaczenie w zależności od kontekstu, w jakim się go użyje, ale ogólnie oznacza: „To coś nie może być zmienione”. Możemy bowiem chcieć uniemożliwić dokonywanie zmian z dwóch względów: projektowania lub wydajności. Ponieważ oba powody różnią się zasadniczo, słowo final może być stosowane niewłaściwie



Podsumowanie

- Oba mechanizmy dziedziczenia i kompozycji pozwalają na stworzenie nowych typów danych z typów już istniejących.
- Mimo silnego nacisku na dziedziczenie w programowaniu obiektowym powinno się preferować kompozycje podczas pierwszego podejścia i stosować dziedziczenie tylko wtedy, gdy jest naprawdę konieczne



koniec

Dziękuję za uwagę

