Podstawy programowania w Javie

Wykład Java - dziedziczenie

dr Agnieszka Zbrzezny

Katedra Metod Matematycznych Informatyki
Wydział Matematyki i Informatyki
Uniwersytet Warmińsko-Mazurski w Olsztynie

02 stycznia 2022

Słowo kluczowe `var`

- Od Javy 10 zmienne lokalne można deklarować za pomocą słowa kluczowego `var` z pominięciem ich typu, jeśli możliwe jest jego określenie na podstawie wartości początkowej.
- Zamiast pisać np. taką deklarację:
  ```
  LocalDate sylwester = LocalDate.of(2020, 12, 31);
  ```
  można napisać:
  ```
  var sylwester = LocalDate.of(2020, 12, 31);
  ```
- W ten sposób można uniknąć zbędnego powtarzania nazwy typu `LocalDate`.

Przykład

- Program `DayOfWeekDemo/DayOfWeekDemo1.java`

Wskazówki dotyczące projektowania klas

1. **Wszystkie pola powinny być prywatne.**
   - To jest najważniejsza ze wszystkich zasad – niestosowanie jej powoduje naruszenie zasad hermetyzacji.
   - Niewykluczone, że z tego powodu będzie konieczne napisanie kilku mutatorów lub metod dostępowych, ale i tak lepiej, aby pola danych pozostały prywatne.
   - Sposób reprezentacji danych może się zmienić, ale sposób ich używania ulega zmianom znacznie rzadziej.
   - Jeżeli dane są prywatne, zmiany w ich reprezentacji nie mają wpływu na użytkowników.
2. **Wszystkie pola powinny jawnie zainicjalizowane.**
   - Java nie inicjuje zmiennych lokalnych, ale zmienne składowe obiektów.
   - Nie należy pozwalać na inicjalizację zmiennych wartościami domyślnymi, tylko inicjalizować je jawnie, podając wartość domyślną lub wartości domyślne we wszystkich konstruktorach.

Wskazówki dotyczące projektowania klas

3. **Należy unikać zbyt wielu typów podstawowych w deklaracji klasy**.
   - Jeśli klasa zawiera kilka powiązanych ze sobą zmiennych tego samego typu, należy je zastąpić nową klasą.
   - Dzięki temu kod klas jest bardziej zrozumiały i łatwiejszy w modyfikacji.
   - Przykładowo, poniższe pola klasy `Customer` można zastąpić nową klasą o nazwie `Address`:

```
private String street;
private String city;
private String state;
private int zip;
```

   - Dzięki temu znacznie prościej jest wprowadzać zmiany w adresach, jak na przykład w przypadku konieczności dodania obsługi adresów międzynarodowych.

Wskazówki dotyczące projektowania klas

4. **Należy używać następującej postaci definicji klasy**:

   ```
   składowe publiczne
   składowe pakietowe
   składowe prywatne
   ```

   a w ramach każdej sekcji zachować kolejność:

   ```
   metody niestatyczne
   metody statyczne
   pola niestatyczne
   pola statyczne
   ```

5. **Należy preferować klasy niezmienne**.
   - Klasa `java.time.LocalDate` jest niezmienna, tzn. żadna metoda nie może zmodyfikować stanu jej obiektu.
   - Dlatego metody takie jak `plusDays` zamiast modyfikować obiekty, zwracają nowe obiekty o zmienionym stanie.

Wskazówki dotyczące projektowania klas

6. **Nazwy klas i metod powinny odzwierciedlać zadania jakie mają do spełnienia.**
   - Podobnie jak zmiennym, klasom należy nadawać nazwy odzwierciedlające ich przeznaczenie (w bibliotece standardowej jest kilka klas, których nazwy budzą wątpliwości, np. klasa `Date`, która opisuje godzinę).
   - Zgodnie z konwencją nazwa klasy powinna być rzeczownikiem (np. `Zamówienie`) lub składać się z przymiotnika i rzeczownika (np. `SzybkieZamówienie`) albo dwóch rzeczowników w odpowiedniej formie (np. `AdresKlienta`).
   - Nazwy akcesorów powinny się zaczynać od pisanego małymi literami słowa `get` (np. `getSalary`), a mutatorów od słowa `set` (np. `setSalary`).

Wskazówki dotyczące projektowania klas

7. **Klasy o zbyt dużej funkcjonalności powinny być dzielone.**
   - Poniższa klasa jest przykładem złego stylu projektowania:

```
public class CardDeck
{
    public CardDeck() { . . .}
    public void shuffle() { . . . }
    public int getTopValue() { . . . }
    public int getTopSuit() { . . . }
    public void draw() { . . . }
    private int[] value;
    private int[] suit;
}
```

   - Klasa ta implementuje dwie odrębne koncepcje: talię kart (`CardDeck`) i związane z nią metody `shuffle` (tasuj) i `draw` (pobierz) oraz metody sprawdzające wartość i kolor karty.

Wskazówki dotyczące projektowania klas

7. **Klasy o zbyt dużej funkcjonalności powinny być dzielone.**
   - Należałoby utworzyć oddzielną klasę o nazwie `Card` reprezentującą kartę.
   - W ten sposób powinny powstać dwie klasy, z których każda ma własny zakres działań:

```
public class Card
{
    public Card(int aValue, int aSuit) { . . . }
    public int getValue() { . . . }
    public int getSuit() { . . . }
    private int value;
    private int suit;
}
```

Wskazówki dotyczące projektowania klas

7. **Klasy o zbyt dużej funkcjonalności powinny być dzielone.**
   - Klasa `CardDeck` implementuje koncepcję talię kart i związane z nią metody `shuffle` (tasuj) i `draw` (pobierz).

```
public class CardDeck
{
    public CardDeck() { . . . }
    public void shuffle() { . . . }
    public Card getTop() { . . . }
    public void draw() { . . . }
    private Card[] cards;
}
```

Wskazówki dotyczące projektowania klas

8. **Nie wszystkie pola muszą posiadać metody `set` i `get`.**
   - Po utworzeniu obiektu zmiany może wymagać na przykład wysokość pensji pracownika, ale z pewnością nie data zatrudnienia.
   - Ponadto obiekty często zawierają składowe, do których nikt spoza klasy nie powinien mieć dostępu.
   - Może to być na przykład tablica skrótów nazw województw w klasie `Address`.
   - Jeżeli metoda typu `get` zwraca wartość referencyjną, a obiekt do którego odnosi się ta referencja jest modyfikowalny, to metoda ta powinna zwrócić referencję do kopii tego obiektu.

Przykład

- Program `BadPoint/TestRectangle.java`
- Program `GoodPoint/TestRectangle.java`

Dziedziczenie – podstawy

- W Javie dziedziczenie jest wyrażane za pomocą słowa **extends**, a cała definicja klasy dziedziczącej po innej klasie schematycznie wygląda następująco:

```
class Potomna extends Bazowa
{
    // wnętrze klasy
}
```

  Zapis taki oznacza, że klasa potomna dziedziczy z klasy bazowej.
- Klasa `Potomna` oprócz własnych składowych będzie posiadała również składowe przejęte z klasy `Bazowa`.
- W Javie klasę pochodną nazywa się też podklasą a klasę bazową nadklasą.

Dziedziczenie - konstruktory

- Podczas tworzenia obiektu klasy potomnej zawsze wywoływany jest konstruktor domyślny klasy bazowej, o ile taki konstruktor istnieje.
- Jeżeli w klasie bazowej nie istnieje konstruktor domyślny, należy jawnie wywołać jeden z pozostałych konstruktorów.
- W tym celu należy zastosować następującą konstrukcję: `super(argumentyKonstruktora)`
- Konstrukcja ze słowem kluczowym **`super`** musi być pierwszą instrukcją konstruktora klasy potomnej.

Przykłady

- Programy `DomyslnyDemo.java`
- Programy `NazwanyPunktDemo.java`

Dziedziczenie - przesłanianie pól i metod

- Kiedy w klasie potomnej znajduje się metoda `f` o takiej samej nazwie i argumentach jak metoda znajdująca się w klasie bazowej występuje zjawisko przesłaniania metod.
- Aby wywołać w klasie potomnej przesłoniętą metodę `f` z klasy bazowej należy użyć konstrukcji: `super.f(argumenty)`
- Kiedy w klasie potomnej znajduje się pole `p` o takiej samej nazwie jak pole znajdujące się w klasie bazowej występuje zjawisko przesłaniania pól.
- Aby w klasie potomnej odwołać się do przesłoniętego pola `p` z klasy bazowej należy użyć konstrukcji: `super.p`

Przykład

- Program `SuperDemo.java`

Dziedziczenie

Klasa `Object`

- W języku Java klasa `Object` jest przodkiem wszystkich klas. Mówiąc inaczej, klasa `Object` jest korzeniem hierarchii klas.
- Wybrane metody klasy `Object`:
  - `public final Class getClass()`
    Zwraca obiekt klasy `Class` zawierający informacje dotyczące klasy obiektu, na rzecz którego wywołano metodę `getClass`.
  - `public boolean equals(Object obj)`
    Porównuje dwa obiekty.
  - `public int hashCode()`
    Zwraca kod mieszający dla obiektu.
  - `public String toString()`
    Zwraca łańcuch reprezentujący wartość obiektu.
  - `protected Object clone()`
    Tworzy kopię obiektu.

Dziedziczenie

Klasa `Object - metoda equals`

- Metoda `equals` z klasy `Object` porównuje dwa obiekty. Jej implementacja w klasie `Object` sprawdza jedynie czy dwie referencje do obiektów są identyczne.
- Dla niektórych klas jest to wystarczające. Często jednak potrzebne jest porównanie stanu obiektów po to, aby uznać, że dwa obiekty są równe, jeżeli mają ten sam stan.
- Wynika stąd potrzeba przedefiniowania metody `equals` w tworzonych klasach.

Dziedziczenie

Metoda `equals`

Specyfikacja języka `Java` wymaga, aby metoda `equals` miała następujące własności:

- zwrotność: `x.equals(x)` zwraca **`true`**.
- symetryczność: `x.equals(y)` zwraca taką samą wartość jak `y.equals(x)`
- przechodniość: jeżeli `x.equals(y)` zwraca **`true`** oraz `y.equals(z)` zwraca **`true`**, to również `x.equals(z)` zwraca **`true`**.
- niezmienność: jeżeli obiekty, do których odnoszą się referencje `x` oraz `y` nie zmieniły się, to kolejne wywołania `x.equals(y)` dają tę samą wartość
- Dla każdej referencyjnej wartości `x` różnej od **`null`**, wywołanie `x.equals(null)` zwraca wartość **`false`**.

Dziedziczenie

Opis tworzenia idealnej metody `equals` dla klasy `K`

1. Nadaj argumentowi metody nazwę `otherObject`.
2. Sprawdź czy referencje **`this`** oraz `otherObject` są identyczne:
   ```
   if (this == otherObject) return true;
   ```
3. Sprawdź, czy referencja `otherObject` jest równa **`null`**:
   ```
   if (otherObject == null) return false;
   ```
4. Porównaj klasy obiektów **`this`** oraz `otherObject`
   - Jeżeli semantyka metody `equals` może zmienić się w podklasach, użyj testu `getClass`:
     ```
     if (getClass() != otherObject.getClass())
       return false;
     ```
   - Jeżeli wszystkie podklasy korzystają z tej samej semantyki można użyć operatora **`instanceof`**:
     ```
     if (!(otherObject instanceof K))
       return false;
     ```

Dziedziczenie

Opis tworzenia idealnej metody `equals` dla klasy `K`

5. Rzutuj obiekt `otherObject` na zmienną typu klasy `K`:
   ```
   K other = (K) otherObject;
   ```
6. Porównaj pola zgodnie z własnymi wymaganiami. Dla pól typów podstawowych użyj operatora `==`, a dla obiektów metody `equals`. Zwróć wartość **`true`** jeżeli pola się zgadzają, lub **`false`** w przeciwnym przypadku.
   Jeżeli przedefiniujesz metodę `equals` w podklasie, użyj odwołania **`super`**`.equals(other)`

Przykład

- Program `EqualsDemo` z pakietu `equals`.