Programowanie obiektowe

Wykład 4: Cechy programowania obiektowego

dr inż. Marcin Luckner
mluckner@mini.pw.edu.pl

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.4
16 marca 2023

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

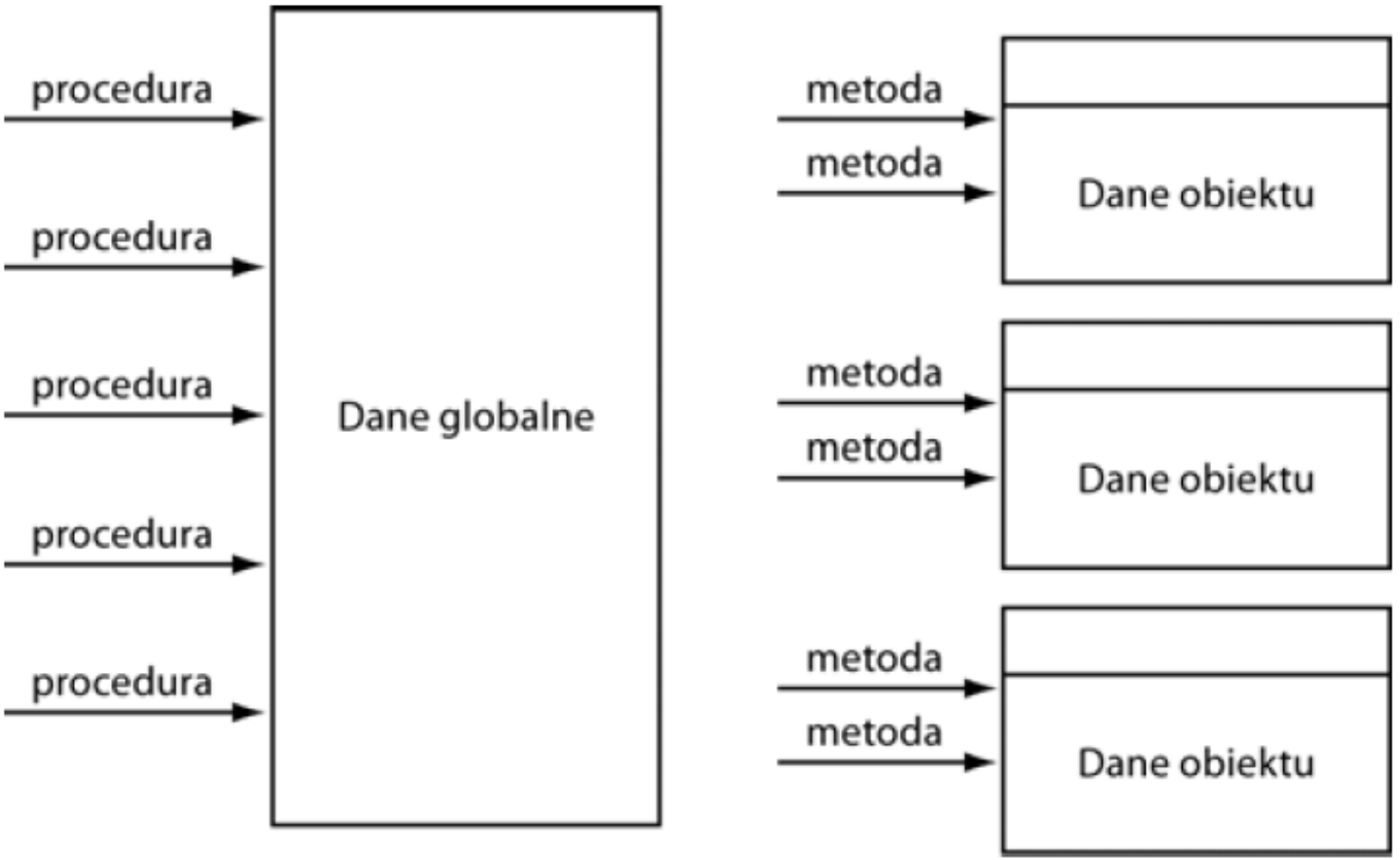
Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Cechy programowania obiektowego

Cechy programowania obiektowego zdefiniowane dla SmallTalka [Kay, 1993].

1. Wszystko jest obiektem.
2. Program jest zbiorem obiektów, komunikujących się poprzez wiadomości.
3. Każdy obiekt jest złożony z innych obiektów.
4. Każdy obiekt posiada typ.
5. Wszystkie obiekty tego samego typu mogą otrzymywać takie same wiadomości.

Programowanie proceduralne a obiektowe



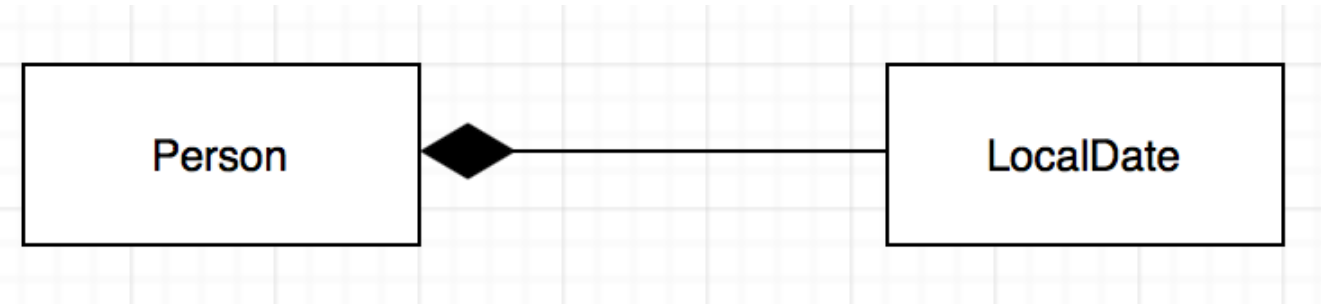
Rysunek 1: Proceduralne a obiektowe podejście do programowania [Horstmann, 2019]

Sens programowania obiektowego

- Stosowanie programowania obiektowego pomaga programistom:
  - Kod staje się czytelniejszy poprzez przypisanie funkcjonalności do obiektów.
  - Ułatwia wielokrotne wykorzystywanie kodu.
  - Współdzielenie kodu odbywa się na poprzez czytelne interfejsy.

Agregacja obiektów

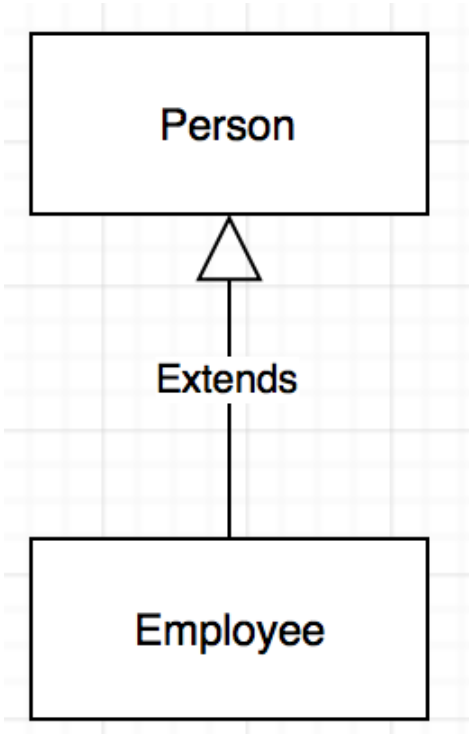
- Dobrze napisany kod chcemy stosować wielokrotnie i tworzenie klas bardzo to ułatwia.
- Nowa klasa może zawierać inne klasy, które zapewniają obsługę pewnych jej aspektów.



Rysunek 2: Klasa `Person` zawiera w sobie klasę `LocalDate`, która odpowiada za zapis dnia urodzin.

Rozszerzanie klas

- Możemy się spotkać z sytuacją, gdy będziemy chcieli zmodyfikować kod, nie tracąc możliwości używania jego starej wersji.
- Jest to możliwe poprzez zdefiniowanie nowej klasy, która *rozszerza* starą.



Rysunek 3: Klasa `Employee` rozszerza klasę `Person`, obiekty z tej klasy są czymś więcej niż osobami.

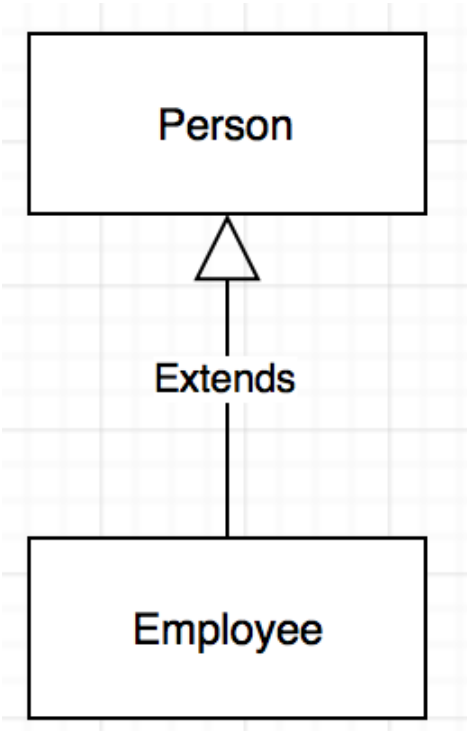
Rozszerzanie klasy Person

Klasa Employee

```
1  public class Employee extends Person {
2
3      Person boss;
4
5      public Employee(String name, LocalDate birthday, Person boss) {
6          super(name, birthday);
7          this.boss = boss;
8      }
9  }
```

- Klasa Employee rozszerza klasę Person poprzez dodanie atrybutu boss.
- Dodaje także konstruktor pozwalający stworzyć obiekt tej klasy.
- Równocześnie zachowuje atrybuty i możliwość wywoływania metod z klasy Person.

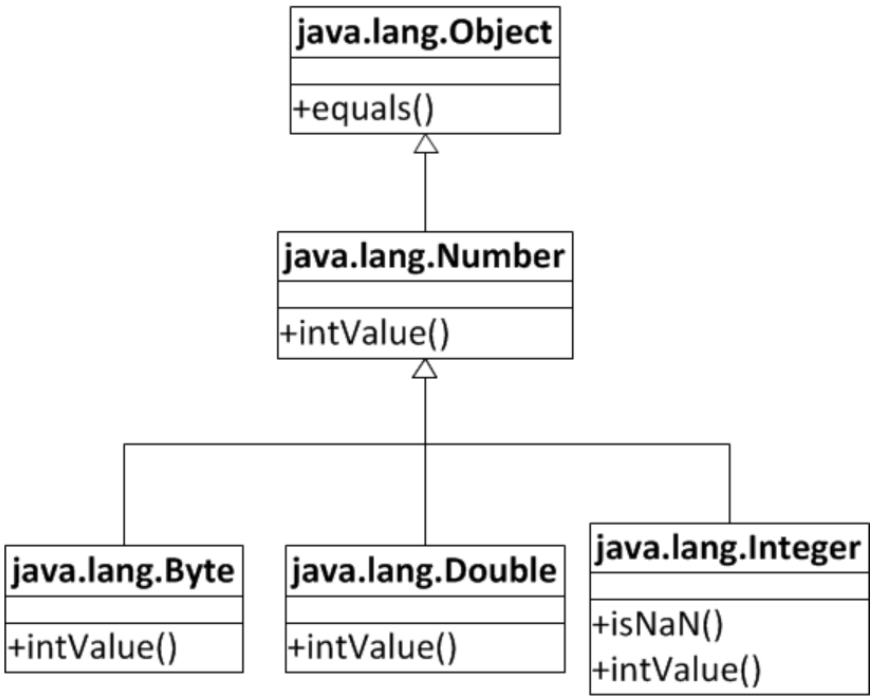
Dziedziczenie



Rysunek 4: Klasa `Employee` dziedziczy po klasie `Person`.

- Klasa `Employee` może wykorzystywać atrybuty i metody klasy `Person`.
- Mówimy, że klasa `Employee` dziedziczy po klasie `Person`.
- Klasa `Employee` jest klasą podrzędną, a klasa `Person` klasą nadrzędną.

Hierarchia klas



Rysunek 5: Hierarchia klas liczbowych

- Klasa `Object` jest klasą nadrzędną dla wszystkich klas w Javie.
- Klasa może mieć kilka klas podrzędnych jak np. klasa `Number`.
- Klasa może mieć tylko jedną klasę nadrzędną.

Przesłanianie

- Dodajmy do klasy `Employee` metodę `toString()`

`toString()`

```
1       @Override
2       public String toString() {
3           return "Employee{boss=" + boss +'}';
4       }
```

- Mamy teraz metodę `toString()` zaimplementowaną w klasie `Person` i `Employee`. Co się stanie, gdy wywołamy tę metodę?

Przesłanianie

```
1   Person batman = new Person("Batman", LocalDate.of(1939, 5, 1));
2   Employee robin
3     = new Employee("Robin", LocalDate.of(1940, 4, 1),batman);
4
5   System.out.println(batman.toString());
6   //Person{name='Batman', birthday=1939-05-01}
7   System.out.println(robin.toString());
8   //Employee{boss=Person{name='Batman', birthday=1939-05-01}}
```

- Metoda `toString()` z klasy `Employee` *przesłania* metodę bazową.

Słowo kluczowe super

- Nadpisywanie rodzi problem uruchamiania metody nadpisanej.
- Z pomocą przychodzi słowo kluczowe super.
- Wywołane jako metoda uruchamia konstruktor klasy bazowej.

Wywołanie konstruktora poprzez super

```
1  public Employee(String name, LocalDate birthday, Person boss) {
2    super(name, birthday);
3    this.boss = boss;
4  }
```

- Wywołane jako atrybut jest referencją do klasy bazowej i pozwala wywoływać jej metody

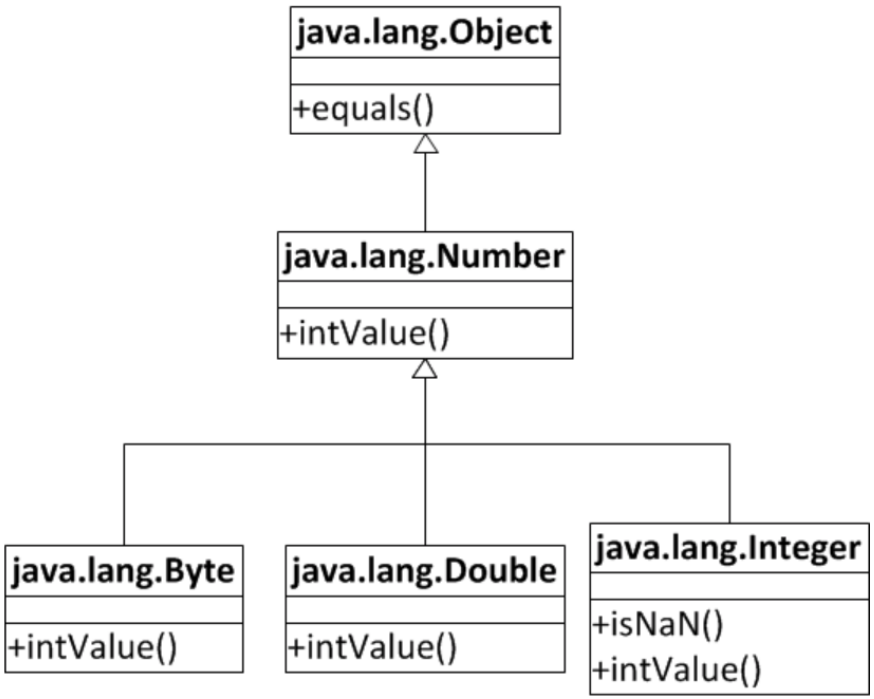
Wywołanie metody poprzez super

```
1  public String whoAmI() {
2    return super.toString();
3  }
4  System.out.println(robin.whoAmI());
5  //Person{name='Robin', birthday=1940-04-01}
```

Polimorfizm

- Obiekt w hierarchii klas możemy traktować jako przynależny do wszystkich typów podrzędnych.
  - Dzięki temu możemy wciąż traktować Pracownika `Employee` jak Osobę `Person`.
- Jednocześnie *wiązanie dynamiczne* powoduje automatyczne wywołanie odpowiedniej implementacji nadpisanych metod (*polimorfizm*).
  - Dlatego wywołanie metody `toString` dopasowane jest do klasy nadrzędnej.

Przykład polimorfizmu



Rysunek 6: Klasy mające wspólne metody

- Każdą z reprezentacji `Byte`, `Double`, `Integer` możemy traktować jako liczbę `Number`.
- Możemy też wykorzystać ich specjalizację np. metody `intValue()`
  - `Integer` zwraca liczbę całkowitą z przedziału $[-2^{31}, 2^{31} - 1]$
  - `Byte` zwraca liczbę całkowitą z przedziału $[-2^{7}, 2^{7} - 1]$
  - `Double` rzutuje liczbę rzeczywistą na całkowitą
  - `(int)(99.9999)); //Zwraca 99`
  - Dla klasy `Number` metoda nie ma określonego działania i nie można jej wywołać (klasa abstrakcyjna).

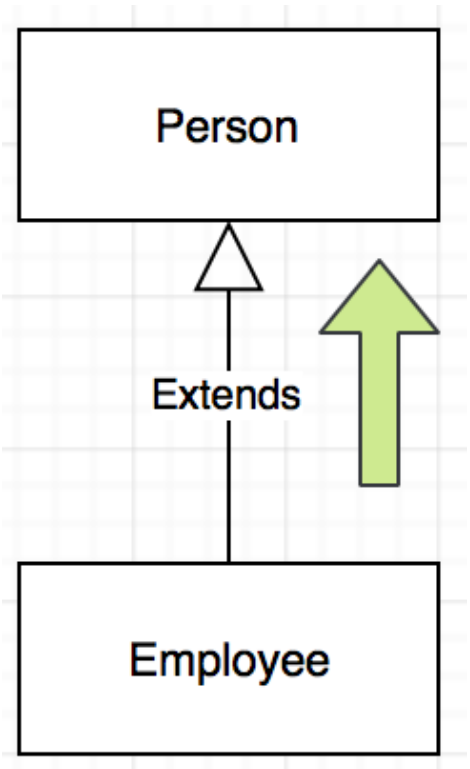
Przesłanianie i modyfikatory

- Przesłanianie ma sens głównie w odniesieniu do metod, ale można nadpisywać też pola.
- Nadpisanie pola statycznego powoduje zmianę jego wartości także w nadrzędnych klasach.
- Możliwość dziedziczenia i nadpisywania klas metod i pól można zablokować stosując modyfikator `final`.

Przynależność obiektu do klasy

- Każdy obiekt może być interpretowany jako przedstawiciel dowolnej klasy bazowej z jego hierarchii.
- Dzięki temu każdy obiekt może być traktowany jako `Object`.
- Można dokonywać konwersji między typami z hierarchii, ale nie jest to proces symetryczny.

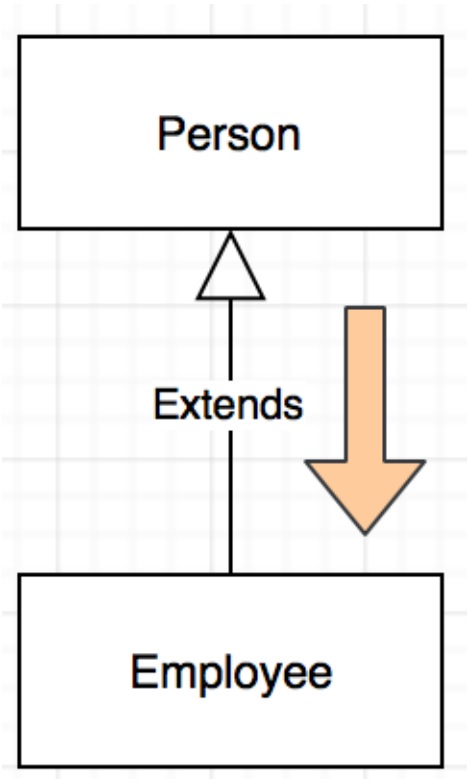
Konwersja rozszerzająca



- Konwersja w górę hierarchii.
- Od klasy podrzędnej do nadrzędnej.
- Zawsze bezpieczna i automatyczna

Rysunek 7: Każdy pracownik jest osobą

Konwersja zawężająca



Rysunek 8: Nie każda osoba jest pracownikiem

- Konwersja w dół hierarchii.
- Od klasy nadrzędnej do podrzędnej.
- Wymaga nadzoru i może powodować błędy.
- Wymaga jawnej deklaracji - *rzutowania*.

Przykład konwersji

- Jak zachowają się poniższe konwersje?.

Konwersja

```
1  Person batman
2    = new Person("Batman", LocalDate.of(1939, 5, 1));
3  Employee robin
4    = new Employee("Robin", LocalDate.of(1940, 4, 1),batman);
5
6  Person person = robin;
7  Employee employee = (Employee)batman;
```

Wynik

```
Exception in thread "main" java.lang.ClassCastException:
 pl.edu.pw.mini.mluckner.op.lecture03.Person
 cannot be cast to
 pl.edu.pw.mini.mluckner.op.lecture03.Employee
```

- Konwersja `Person person = robin;` zadziałała poprawnie, wyjątek spowodowała kolejna konwersja.

Tryb deklarowany i aktualny

- W poniższym przypadku mamy do czynienia z referencją `Person` do obiektu z klasy `Employee`.

```
1   Person person
2     = new Employee("Robin", LocalDate.of(1940, 4, 1),batman);
```

- Obiekt `person` będzie miał teraz dwa typy
  - Typ deklarowany `Person`
  - Typ aktualny `Employee`
- W różny sposób będziemy mogli odwoływać się do metod i atrybutów tego obiektu

Dostęp do metod i argumentów

- Metody
  - Metody statyczne są wywoływane dla typu deklarowanego.
  - Metody niestatyczne są wywoływane dla typu aktualnego.
- Argumenty
  - Zarówno statyczne jak i niestatyczne pola są odczytywane z typu deklarowanego.

Jekyll & Hyde

```
1   public class Jekyll{
2
3   static int intelligence = 180;
4   int height = 170;
5
6   static String greeting(){
7    return "Good day Sir";
8   }
9
10  String name(){
11    return "dr. Henry Jekyll";
12  }
```

```
1   public class Hyde extends Jekyll{
2
3   static int intelligence = 90;
4   int height = 200;
5
6   static String greeting(){
7    return "Hey man";
8   }
9
10  String name(){
11    return "Edward Hyde";
12  }
```

```
1   Jekyll jekyll= new Hyde();
2   System.out.println(jekyll.greeting()+", my name is "+jekyll.name());
3   System.out.println("intelligence: "+jekyll.intelligence+"height:
        "+jekyll.height);
```

```
Good day Sir, my name is Edward Hyde
intelligence: 180 height: 170
```

Enkapsulacja

- Z punktu widzenia programisty obiekty powinny być podstawowymi elementami budowy kodu.
- Ich znajomość ogranicza się do znajomości interfejsu, który udostępniają.
- Budując obiekty ukrywamy sposób implementacji metod przed jej użytkownikami.
- Proces ten nazywamy *enkapsulacją*. Pozwala on na:
  - Swobodną zmianę sposobu implementacji metod, o ile nie zmienimy samego interfejsu.
  - Zapewnienie bezpieczeństwa i prywatności danych znajdujących się wewnątrz obiektów.

Ograniczanie dostępu do atrybutów

Klasa `NiceGirl`

```
1  public class NiceGirl {
2      private String diary;
3      protected String facebook;
4      public String blog;
5  }
```

private Tylko obiekty danej klasy mają dostęp do tej zmiennej.

protected Dostęp także dla obiektów z klas pochodnych.

public Dostęp dla wszystkich obiektów

Domyślnie atrybuty są dostępne tylko dla kas, które znajduje się w tym samym pakiecie.

Zastosowanie enkapsulacji

Klasa ReasonableNiceGirl

```
1  public class ReasonableNiceGirl {
2
3      private String diary;
4      private String facebook;
5      private String blog;
6
7      protected void
           sendMessageToFacebook(String
           message) {
8          //Some code that modifies
               facebook;
9      }
10
11     public void
           addCommentToBlog(String
           comment) {
12         //Some code that moderates
               and add comments;
13     }
14
15 }
```

- Nowy obiekt ukrywa wszystkie zmienne, stają się one prywatne.
- Udostępnia tylko zbiór publicznych metod (interfejs) do komunikacji z innymi obiektami.
- Kontrolujemy dostęp do zmiennych, sposób ich modyfikacji i moderujemy działania innych obiektów.

Dostęp do atrybutów

- Dobre praktyki, a także niektóre zastosowania, wymagają zapewnienia odpowiednich metod dostępu do atrybutów.

Metoda odczytu getter

```
1  TypAtrybutu getNazwaAtrybutu();
2  Boolean isNazwaAtrybutu(); //Tylko dla zmiennych logicznych
```

Metoda zapisu setter

```
1  void setNazwaAtrybutu(TypAtrybutu wartoscAtrybutu)
```

Interfejs `interface`

- Pokłosiem wprowadzenia enkapsulacji jest ograniczenie postrzegania obiektu do jego interfejsu.
- W Javie przyjmuje to postać sformalizowaną i możemy definiować `interface` jako zbiór metod, które implementuje obiekt.
- Zagadnienie interfejsu jest na tyle istotne, że poświęcimy mu osobny wykład.

Bibliografia

[Horstmann, 2019] Horstmann, C. S. (2019).
Java. Podstawy. Wydanie XI.
Helion.

[Kay, 1993] Kay, A. C. (1993).
The Early History of Smalltalk.
SIGPLAN Not., 28(3):69–95.

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.