Programowanie w Java Dziedziczenie

Łukasz Rybka · Gdańsk 2016/17

SOLID

SOLID

SRP - Single responsibility principle

SOLID

SRP - Single responsibility principle

OCP - Open/closed principle

SOLID

SRP - Single responsibility principle

OCP - Open/closed principle

LSP - Liskov substitution principle

SOLID

SRP - Single responsibility principle

OCP - Open/closed principle

LSP - Liskov substitution principle

ISP - Interface segregation principle

SOLID

SRP - Single responsibility principle

OCP - Open/closed principle

LSP - Liskov substitution principle

ISP - Interface segregation principle

DIP - Dependency inversion principle

Open/closed principle

> *Software entities … should be open for extension, but closed for modification.*
>
> — Wikipedia

Czy klasy nie są wystarczające?

Czy klasy nie są wystarczające?

są skuteczną metodą dekompozycji (np. krzesło vs oparcie + siedzisko + nogi)

Czy klasy nie są wystarczające?

są skuteczną metodą dekompozycji (np. krzesło vs oparcie + siedzisko + nogi)

bardzo elastyczne

Czy klasy nie są wystarczające?

są skuteczną metodą dekompozycji (np. krzesło vs oparcie + siedzisko + nogi)

bardzo elastyczne

łatwa do opisania semantyka za pomocą asercji

Czy klasy nie są wystarczające?

są skuteczną metodą dekompozycji (np. krzesło vs oparcie + siedzisko + nogi)

bardzo elastyczne

łatwa do opisania semantyka za pomocą asercji

rozdzielność interfejsu od implementacji

Podstawowa konwencja i terminologia

Podstawowa konwencja i terminologia

POTOMKIEM KLASY C:

Podstawowa konwencja i terminologia

POTOMKIEM KLASY C:

jest dowolna klasa, która dziedziczy bezpośrednio lub pośrednio po C, łącznie z samą klasą C).

Podstawowa konwencja i terminologia

POTOMKIEM KLASY C:

jest dowolna klasa, która dziedziczy bezpośrednio lub pośrednio po C, łącznie z samą klasą C).

WŁAŚCIWY POTOMEK C:

Podstawowa konwencja i terminologia

POTOMKIEM KLASY C:

jest dowolna klasa, która dziedziczy bezpośrednio lub pośrednio po C, łącznie z samą klasą C).

WŁAŚCIWY POTOMEK C:

to potomek tej klasy inny niż C.

Podstawowa konwencja i terminologia

POTOMKIEM KLASY C:

jest dowolna klasa, która dziedziczy bezpośrednio lub pośrednio po C, łącznie z samą klasą C).

WŁAŚCIWY POTOMEK C:

to potomek tej klasy inny niż C.

PRZODKIEM C:

Podstawowa konwencja i terminologia

POTOMKIEM KLASY C:

jest dowolna klasa, która dziedziczy bezpośrednio lub pośrednio po C, łącznie z samą klasą C).

WŁAŚCIWY POTOMEK C:

to potomek tej klasy inny niż C.

PRZODKIEM C:

jest taka klasa A, że C jest potomkiem C. Właściwy przodek C to taka klasa A, że C jest właściwym potomkiem A.

Podstawowa zasada dziedziczenia w Javie

> *Dziedziczenie jest używane zawsze, kiedy tworzymy jakąś klasę (…) automatycznie dziedziczymy ze standardowej (…) klasy bazowej (…) Object.*
>
> — Bruce Eckel

Klasa bazowa

```
package pl.org.dragonia.oopl;

public class Animal {
    public void sleep() {
        System.out.println("An animal sleeps...");
    }

    public void eat() {
        System.out.println("An animal eats...");
    }
}
```

Klasy pochodne

```
public class Bird extends Animal {(1)
        @Override(2)
        public void sleep() {
                System.out.println("A bird sleeps...");
        }

        @Override(2)
        public void eat() {
                System.out.println("A bird eats...");
        }
}
```

(1) Klasa Bird **rozszerza** klasę Animal

(2) metody sleep() oraz eat() są **nadpisywane** w klasie Bird

Klasy pochodne

```
public class Dog extends Animal {
        @Override
        public void sleep() {
                System.out.println("A dog sleeps...");
        }

        @Override
        public void eat() {
                System.out.println("A dog eats...");
        }
}
```

Wykorzystanie dziedziczenia

```
public class AnimalsInheritance {

        public static void main(String[] agrs) {

        Animal animal = new Animal();
        animal.sleep();
        animal.eat();

        Bird bird = new Bird();
        bird.sleep();
        bird.eat();

        Dog dog = new Dog();
        dog.sleep();
        dog.eat();
    }
}
```

Konstruktory w kontekście dziedziczenia

```
class Art {
    Art() {
        System.out.println("Art class constructor");
    }
}
```

Konstruktory w kontekście dziedziczenia

```
class Art {
    Art() {
        System.out.println("Art class constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing class constructor");
    }
}
```

Konstruktory w kontekście dziedziczenia

```
class Art {
    Art() {
        System.out.println("Art class constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing class constructor");
    }
}
```

```
public class Cartoon extends Drawing {
    public Cartoon() {
        System.out.println("Cartoon class constructor");
    }

    public static void main(String[] args) {
        Art art = new Art();
        Drawing drawing = new Drawing();
        Cartoon cartoon = new Cartoon();
    }
}
```

Słowo kluczowe super

Słowo kluczowe super

W zależności od kontekstu ma inne zastosowanie

Słowo kluczowe super

W zależności od kontekstu ma inne zastosowanie

Wywołane w konstruktorze - wywołuje konstruktor klasy bazowej

Słowo kluczowe super

W zależności od kontekstu ma inne zastosowanie

Wywołane w konstruktorze - wywołuje konstruktor klasy bazowej

Wywołane w metodzie - wywołuje metodę klasy bazowej z zadaną nazwą

Słowo kluczowe super

W zależności od kontekstu ma inne zastosowanie

Wywołane w konstruktorze - wywołuje konstruktor klasy bazowej

Wywołane w metodzie - wywołuje metodę klasy bazowej z zadaną nazwą

Daje dostęp tylko do bezpośredniego właściwego przodka

Użycie super() w konstruktorze

```
class Art {
    Art() {
        System.out.println("Art class constructor");
    }

    Art(String painter) {
        System.out.println("Art painter: " + painter);
    }
}

public class Drawing extends Art {
    public Drawing() {
        System.out.println("Cartoon class constructor");
    }

    public Drawing(String painter) {
        super(painter);
        System.out.println("Cartoon painter: " + painter);
    }

    public static void main(String[] args) {
        Art art = new Art();
        Drawing drawing = new Drawing("Vincent van Gogh");
    }
}
```

Użycie super() w konstruktorze - pułapka

```
class Art {
    Art(String painter) {
        System.out.println("Art painter: " + painter);
    }
}

public class Drawing extends Art {
    public Drawing(String painter) {
        super();
        System.out.println("Cartoon painter: " + painter);
    }
    public static void main(String[] args) {
        Drawing drawing = new Drawing("Vincent van Gogh");
    }
}
```

Użycie super w metodzie

```
class Animal {
    public void sleep() {
        System.out.println("An animal sleeps...");
    }
}

public class Bird extends Animal {
    @Override
    public void sleep() {
        System.out.println("A bird sleeps...");
        super.sleep();
    }

    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.sleep();
    }
}
```

Modyfikatory dostępu

Modyfikatory dostępu

PUBLIC:

Modyfikatory dostępu

PUBLIC:

pozwala na dostęp wszystkich klasom z dowolnego pakietu

Modyfikatory dostępu

PUBLIC:

pozwala na dostęp wszystkich klasom z dowolnego pakietu

PACKAGE (DOMYŚLNY):

Modyfikatory dostępu

PUBLIC:

pozwala na dostęp wszystkich klasom z dowolnego pakietu

PACKAGE (DOMYŚLNY):

dostęp do danej klasy/metody/pola mają jedynie klasy z tego samego pakietu

Modyfikatory dostępu

Modyfikatory dostępu

PRIVATE:

Modyfikatory dostępu

PRIVATE:

nikt poza samą klasą nie ma dostępu do danej klasy/pola

Modyfikatory dostępu

PRIVATE:

nikt poza samą klasą nie ma dostępu do danej klasy/pola

PROTECTED:

Modyfikatory dostępu

PRIVATE:

nikt poza samą klasą nie ma dostępu do danej klasy/pola

PROTECTED:

dostęp do metody/pola jedynie poprzez dziedziczenie

Modyfikator dostępu "protected"

Modyfikator dostępu "protected"

Jeżeli dziedziczymy po klasie z tego samego pakietu - mamy dostęp do jej publicznych i pakietowych składowych

Modyfikator dostępu "protected"

Jeżeli dziedziczymy po klasie z tego samego pakietu - mamy dostęp do jej publicznych i pakietowych składowych

Jeżeli dziedziczymy po klasie z innego pakietu - mamy dostęp jedynie do publicznych składowych

Modyfikator dostępu "protected"

Jeżeli dziedziczymy po klasie z tego samego pakietu - mamy dostęp do jej publicznych i pakietowych składowych

Jeżeli dziedziczymy po klasie z innego pakietu - mamy dostęp jedynie do publicznych składowych

Modyfikator protected umożliwia dostęp przez dziedziczenie poza pakietem

Rzutowanie

Rzutowanie

Istnieją dwa typu rzutowania: upcasting (w górę) oraz downcasting (w dół)

Rzutowanie

Istnieją dwa typu rzutowania: upcasting (w górę) oraz downcasting (w dół)

Upcasting: zawsze bezpieczne, od szczegółu (potomka) do ogółu (przodka)

Rzutowanie

Istnieją dwa typu rzutowania: upcasting (w górę) oraz downcasting (w dół)

Upcasting: zawsze bezpieczne, od szczegółu (potomka) do ogółu (przodka)

Rzutować w górę możemy aż do pierwszego przodka (klasy Object)

Rzutowanie

Istnieją dwa typu rzutowania: upcasting (w górę) oraz downcasting (w dół)

Upcasting: zawsze bezpieczne, od szczegółu (potomka) do ogółu (przodka)

Rzutować w górę możemy aż do pierwszego przodka (klasy Object)

Rzutowaniem w górę nie uzyskamy dostępu do metod przodka!

Upcasting a wywołanie metod

```
class Art {
    public void pain() {
        System.out.println("Art...");
    }
}

class Drawing extends Art {
    @Override
    public void pain() {
        System.out.println("Drawing...");
    }

    public static void main(String[] args) {
        Drawing drawing = new Drawing();

        drawing.pain();❶
        ((Art) drawing).pain();❷
    }
}
```

❶ Wypisze na ekranie "Drawing..."

❷ Wypisze na ekranie ???

Upcasting a wywołanie metod

```
class Art {
    public void pain() {
        System.out.println("Art...");
    }
}

class Drawing extends Art {
    @Override
    public void pain() {
        System.out.println("Drawing...");
    }

    public static void main(String[] args) {
        Drawing drawing = new Drawing();

        drawing.pain();❶
        ((Art) drawing).pain();❷
    }
}
```

❶ Wypisze na ekranie "Drawing..."

❷ Wypisze na ekranie "Drawing..."

Sprawdzanie typu obiektu

Sprawdzanie typu obiektu

Każda klasa dziedziczy po klasie Object, która posiada metodę getClass()

Sprawdzanie typu obiektu

Każda klasa dziedziczy po klasie Object, która posiada metodę getClass()

Każda klasa C dziedzicząca po klasie A jest także typu A

Sprawdzanie typu obiektu

Każda klasa dziedziczy po klasie Object, która posiada metodę getClass()

Każda klasa C dziedzicząca po klasie A jest także typu A

Operator instanceof mówi nam, czy obiekt jest danego typu (z uwzględnieniem dziedziczenia)

Sprawdzanie typu obiektu

```
package pl.org.dragonia.oopl;

class Animal {
    // ...
}

class Bird extends Animal {
    // ...
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Bird bird = new Bird();

        System.out.println(animal.getClass().getName());
        System.out.println(animal instanceof Animal);
        System.out.println(animal instanceof Animal);

        System.out.println(bird.getClass().getName());
        System.out.println(bird instanceof Animal);
        System.out.println(bird instanceof Animal);
    }
}
```

Klasy abstrakcyjne

Klasy abstrakcyjne

Nie można tworzyć obiektów klasy abstrakcyjnej

Klasy abstrakcyjne

Nie można tworzyć obiektów klasy abstrakcyjnej

Metoda abstrakcyjna nie może posiadać implementacji

Klasy abstrakcyjne

Nie można tworzyć obiektów klasy abstrakcyjnej

Metoda abstrakcyjna nie może posiadać implementacji

Klasa z przynajmniej jedną metodą abstrakcyjną musi być abstrakcyjna

Klasy abstrakcyjne

```
package pl.org.dragonia.oopl;

abstract class Animal {
    public void sleep();
}

class Bird extends Animal {
    @Override
    public void sleep() {
        System.out.println("An animal sleeps...");
    }
}

public class Main {
    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.sleep();
    }
}
```

Polimorfizm

> *Polimorfizm oznacza zdolność przyjmowania różnych postaci.*
>
> — Bertrand Meyer

Przypisania polimorficzne

> *(…) przypisania, w których typ źródła jest inny niż typ celu są nazywane* ***przypisaniami polimorficznymi*****.**
>
> — Bertrand Meyer

Polimorficzne struktury danych

> *(…) struktury danych, zawierające obiekty różnych typów są nazywane* ***polimorficznymi strukturami danych*****.**
>
> — Bertrand Meyer

Przykład wiązania polimorficznego

```
package pl.org.dragonia.oopl;

class Animal {
    public void sleep() {
        System.out.println("Private sleep method");
    }
}

class Bird extends Animal {
    @Override
    public void sleep() {
        System.out.println("An bird sleeps...");
    }

    public static void main(String[] args) {
        Animal animal = new Bird();
        animal.sleep();
    }
}
```

Wiązanie polimorficzne w argumencie metody

```
package pl.org.dragonia.oopl;

abstract class Animal {
    public abstract void sleep();

    public static void sleep(Animal animal) {
        animal.sleep();
    }
}

class Bird extends Animal {
    @Override
    public void sleep() {
        System.out.println("An bird sleeps...");
    }

    public static void main(String[] args) {
        Animal animal = new Bird();
        Animal.sleep(animal);
    }
}
```

Wiązanie

> *Połączenie wywołania metody z jej ciałem nazywamy **wiązaniem** (ang. **binding**).*
>
> — Bruce Eckel

Rodzaje wiązań w Javie

Rodzaje wiązań w Javie

WCZESNE WIĄZANIE (EARLY BINDING):

Rodzaje wiązań w Javie

WCZESNE WIĄZANIE (EARLY BINDING):

Dokojuje się przed wykonaniem programu (wykonywane przez kompilator oraz linker)

Rodzaje wiązań w Javie

WCZESNE WIĄZANIE (EARLY BINDING):

Dokojuje się przed wykonaniem programu (wykonywane przez kompilator oraz linker)

PÓŹNE WIĄZANIE (LATE BINDING):

Rodzaje wiązań w Javie

WCZESNE WIĄZANIE (EARLY BINDING):

Dokojuje się przed wykonaniem programu (wykonywane przez kompilator oraz linker)

PÓŹNE WIĄZANIE (LATE BINDING):

Odbywa się w czasie wykonania programu i opiera się na właściwym typie obiektu

Wiązanie w Javie

> *Wszelkie wiązania w Javie są wiązaniami późnymi, chyba że metoda została zadeklarowana z użyciem modyfikatora final.*
>
> — Bruce Eckel

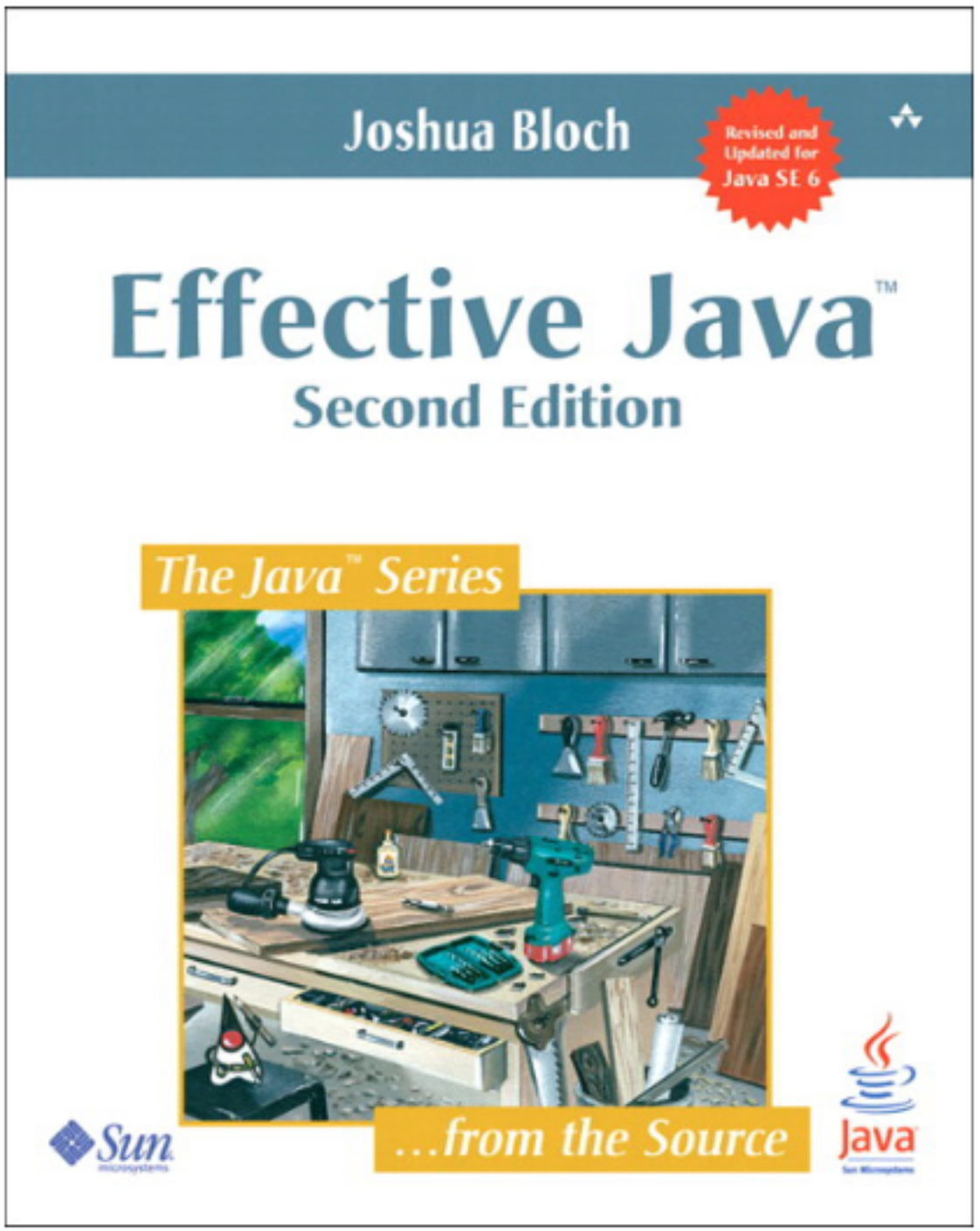
Przesłanianie metod prywatnych

```
package pl.org.dragonia.oopl;

class Animal {
    private void sleep() {
        System.out.println("Private sleep method");
    }

    public static void main(String[] args) {
        Animal animal = new Bird();
        animal.sleep();
    }
}

class Bird extends Animal {
    public void sleep() {
        System.out.println("An bird sleeps...");
    }
}
```

Metody statyczne a polimorfizm

```
class Super {
    public static String staticGet() {
        return "Bazowa wersja staticGet()";
    }
    public String dynamicGet() {
        return "Bazowa wersja dynamicGet()";
    }
}
class Sub extends Super {
    public static String staticGet() {
        return "Pochodna wersja staticGet()";
    }
    public String dynamicGet() {
        return "Pochodna wersja dynamicGet()";
    }

    public static void main(String[] args) {
        Super sup = new Sub();
        System.out.println(sup.staticGet());
        System.out.println(sup.dynamicGet());
    }
}
```

Effective Java



Item10: Always override toString

> *(…) providing a good toString implementation makes your class much more pleasant to use.*
>
> — Joshua Bloch

Item10: Always override toString

> *When practical, the toString method should return **all** of the interesting information contained in the object (…).*
>
> — Joshua Bloch

ANY QUESTIONS?