Java

Deklaracje i prawa dostępu

Piotr Fulmański

WydziaĹ, Matematyki i Informatyki,
Uniwersytet Łódzki, Polska

December 4, 2010

Spis treści

Java – krótka charakterystyka

Można powiedzieć, iż każdy program napisany w Javie jest zbiorem pewnych **obiektów** (ang. *objects*). Komunikacja między obiektami zazwyczaj odbywa się za pośrednictwem **metod** (ang. *methods*). To czym jest obiekt (jakiego jest rodzaju, typu, co potrafi) definiowane jest przez **klasę** (ang. *class*) lub **interfejs** (ang. *interface*). Mówiąc trochę bardziej precyzyjnie

Java – krótka charakterystyka

Obiekt, klasa, . . .

- **Klasa** Jest to pewnego rodzaju wzorzec opisujący (definiujący) **stany** jakie może przyjąć każdy obiekt utworzony w oparciu o daną klasę a także **zachowania** (czynności, akcje) jakie są dla niego charakterystyczne (czynności jakie może wykonać, akcję jakie można „zlecić” jemu do wykonania).
- **Obiekt**
- **Stan**
- **Zachowanie**

Obiekt, klasa, . . .

- **Klasa** Jest to pewnego rodzaju wzorzec opisujący (definiujący) **stany** jakie może przyjąć każdy obiekt utworzony w oparciu o daną klasę a także **zachowania** (czynności, akcje) jakie są dla niego charakterystyczne (czynności jakie może wykonać, akcję jakie można „zlecić” jemu do wykonania).
- Obiekt
- Stan
- Zachowanie

Obiekt, klasa, . . .

- **Klasa** Jest to pewnego rodzaju wzorzec opisujący (definiujący) **stany** jakie może przyjąć każdy obiekt utworzony w oparciu o daną klasę a także **zachowania** (czynności, akcje) jakie są dla niego charakterystyczne (czynności jakie może wykonać, akcję jakie można „zlecić" jemu do wykonania).
- **Obiekt**
- Stan
- Zachowanie

Obiekt, klasa, . . .

- **Klasa** Jest to pewnego rodzaju wzorzec opisujący (definiujący) **stany** jakie może przyjąć każdy obiekt utworzony w oparciu o daną klasę a także **zachowania** (czynności, akcje) jakie są dla niego charakterystyczne (czynności jakie może wykonać, akcję jakie można „zlecić” jemu do wykonania).
- **Obiekt**
- **Stan**
- **Zachowanie**

Obiekt, klasa, . . .

- **Klasa** Jest to pewnego rodzaju wzorzec opisujący (definiujący) **stany** jakie może przyjąć każdy obiekt utworzony w oparciu o daną klasę a także **zachowania** (czynności, akcje) jakie są dla niego charakterystyczne (czynności jakie może wykonać, akcję jakie można „zlecić" jemu do wykonania).
- **Obiekt**
- **Stan**
- **Zachowanie**

Dziedziczenie

Pakiet jako zbiór klas

Nazwy i słowa kluczowe

Zwykle zagadnienie nazewnictwa stosowanego w Javie bywa marginalizowane – przecież to tylko nazwy. Z nazwami jest jednak tak jak z wymową – przyjemniej rozmawia się nam z kimś o dobrej dykcji niż osobą mamroczącą coś niewyraźnie pod nosem. Dodatkowo niewyraźna wymowa bywa często źródłem pomyłek i przyczyną braku wzajemnego zrozumienia. Tak samo dzieje się też w programowaniu. Dlatego postanowiliśmy poświęcić nazwom osobny podrozdział w dalszej części podręcznika (patrz

Nazwy

Mówiąc o nazwach, pamętać musimy, że podlegają one (jednocześnie) trzem konwencjom nazewniczym.

Identyfikatory poprawne składniowo

Bez względu na to jaką nazwę (identyfikator)sobie wymyślimy, czy stosujemy się do własnej konwencji czy narzuconej np. przez firmę, w której pracujemy, musi ona być poprawna składniowo, czyli być zbudowana zgodnie z zasadami według jakich kompilator będzie ją w przyszłości sprawdzał.

Identyfikatory poprawne składniowo

- Dopuszczalne znaki
- Istotna wielkość liter
- Pierwszy znak
- Długość identyfikatora
- Słowa kluczowe

Identyfikatory poprawne składniowo

- Dopuszczalne znaki
- Istotna wielkość liter
- Pierwszy znak
- Długość identyfikatora
- Słowa kluczowe

Identyfikatory poprawne składniowo

- Dopuszczalne znaki
- Istotna wielkość liter
- Pierwszy znak
- Długość identyfikatora
- Słowa kluczowe

Identyfikatory poprawne składniowo

- Dopuszczalne znaki
- Istotna wielkość liter
- Pierwszy znak
- Długość identyfikatora
- Słowa kluczowe

Identyfikatory poprawne składniowo

- Dopuszczalne znaki
- Istotna wielkość liter
- Pierwszy znak
- Długość identyfikatora
- Słowa kluczowe

Identyfikatory poprawne składniowo

- Dopuszczalne znaki
- Istotna wielkość liter
- Pierwszy znak
- Długość identyfikatora
- Słowa kluczowe

Słowa kluczowe

tabela

Konwencja nazewnicza firmy Sun

Java Code Convention, September 12, 1997

`http://www.oracle.com/technetwork/java/codeconv-138413.html`

Konwencja nazewnicza firmy Sun

- Klasy i interfejsy
- Metody
- Zmienne
- Stałe

Konwencja nazewnicza firmy Sun

- Klasy i interfejsy
- Metody
- Zmienne
- Stałe

Konwencja nazewnicza firmy Sun

- Klasy i interfejsy
- Metody
- Zmienne
- Stałe

Konwencja nazewnicza firmy Sun

- Klasy i interfejsy
- Metody
- Zmienne
- Stałe

Konwencja nazewnicza firmy Sun

- Klasy i interfejsy
- Metody
- Zmienne
- Stałe

Konwencja nazewnicza JavaBean

Co to jest JavaBean?

Konwencja nazewnicza własności (ang. *property*) JavaBean

- Gettery
- Settery

Konwencja nazewnicza własności (ang. *property*) JavaBean

- Gettery
- Settery

Konwencja nazewnicza własności (ang. *property*) JavaBean

- Gettery
- Settery

Konwencja nazewnicza (ang. *listener*) JavaBean

- rejestrowanie listenera (add)
- wyrejestrowanie listenera (remove)
- wspólne: końcówka to `Listener`

Konwencja nazewnicza (ang. *listener*) JavaBean

- rejestrowanie listenera (add)
- wyrejestrowanie listenera (remove)
- wspólne: końcówka to `Listener`

Konwencja nazewnicza (ang. *listener*) JavaBean

- rejestrowanie listenera (add)
- wyrejestrowanie listenera (remove)
- wspólne: końcówka to `Listener`

Konwencja nazewnicza (ang. *listener*) JavaBean

- rejestrowanie listenera (add)
- wyrejestrowanie listenera (remove)
- wspólne: końcówka to `Listener`

Wiadomości ogólne

- Wpływ komentarzy
- Tylko jedna klasa publiczna w pliku źródłowym (nie ma limitu na klasy niepubliczne).
- Class name and file name matching rule (tylko gdy istnieje klasa publiczna).
- Umiejscowienie słowa kluczowego `package` w klasie.
- Umiejscowienie słowa kluczowego `import` w klasie.
- Globalne działanie `package` i `import` w stosunku do klas umieszczonych w tym samym pliku.

Wiadomości ogólne

- Wpływ komentarzy
- Tylko jedna klasa publiczna w pliku źródłowym (nie ma limitu na klasy niepubliczne).
- Class name and file name matching rule (tylko gdy istnieje klasa publiczna).
- Umiejscowienie słowa kluczowego `package` w klasie.
- Umiejscowienie słowa kluczowego `import` w klasie.
- Globalne działanie `package` i `import` w stosunku do klas umieszczonych w tym samym pliku.

Wiadomości ogólne

- Wpływ komentarzy
- Tylko jedna klasa publiczna w pliku źródłowym (nie ma limitu na klasy niepubliczne).
- Class name and file name matching rule (tylko gdy istnieje klasa publiczna).
- Umiejscowienie słowa kluczowego `package` w klasie.
- Umiejscowienie słowa kluczowego `import` w klasie.
- Globalne działanie `package` i `import` w stosunku do klas umieszczonych w tym samym pliku.

Wiadomości ogólne

- Wpływ komentarzy
- Tylko jedna klasa publiczna w pliku źródłowym (nie ma limitu na klasy niepubliczne).
- Class name and file name matching rule (tylko gdy istnieje klasa publiczna).
- Umiejscowienie słowa kluczowego `package` w klasie.
- Umiejscowienie słowa kluczowego `import` w klasie.
- Globalne działanie `package` i `import` w stosunku do klas umieszczonych w tym samym pliku.

Wiadomości ogólne

- Wpływ komentarzy
- Tylko jedna klasa publiczna w pliku źródłowym (nie ma limitu na klasy niepubliczne).
- Class name and file name matching rule (tylko gdy istnieje klasa publiczna).
- Umiejscowienie słowa kluczowego `package` w klasie.
- Umiejscowienie słowa kluczowego `import` w klasie.
- Globalne działanie `package` i `import` w stosunku do klas umieszczonych w tym samym pliku.

Wiadomości ogólne

- Wpływ komentarzy
- Tylko jedna klasa publiczna w pliku źródłowym (nie ma limitu na klasy niepubliczne).
- Class name and file name matching rule (tylko gdy istnieje klasa publiczna).
- Umiejscowienie słowa kluczowego `package` w klasie.
- Umiejscowienie słowa kluczowego `import` w klasie.
- Globalne działanie `package` i `import` w stosunku do klas umieszczonych w tym samym pliku.

Wiadomości ogólne

- Wpływ komentarzy
- Tylko jedna klasa publiczna w pliku źródłowym (nie ma limitu na klasy niepubliczne).
- Class name and file name matching rule (tylko gdy istnieje klasa publiczna).
- Umiejscowienie słowa kluczowego `package` w klasie.
- Umiejscowienie słowa kluczowego `import` w klasie.
- Globalne działanie `package` i `import` w stosunku do klas umieszczonych w tym samym pliku.

Deklarowanie klas

Najprostsza deklaracja klasy

```
class myClass { }
```

Modyfikatory

- Modyfikatory dostępu: `public`, `protected`, `private`.
- Inne modyfikatory: `final`, `abstract`, `strictfp`

Deklarowanie klas

Najprostsza deklaracja klasy

```
class myClass { }
```

Modyfikatory

- Modyfikatory dostępu: `public`, `protected`, `private`.
- Inne modyfikatory: `final`, `abstract`, `strictfp`

Deklarowanie klas

Najprostsza deklaracja klasy

```
class myClass { }
```

Modyfikatory

- Modyfikatory dostępu: `public`, `protected`, `private`.
- Inne modyfikatory: `final`, `abstract`, `strictfp`

Deklarowanie klas

Najprostsza deklaracja klasy

```
class myClass { }
```

Modyfikatory

- Modyfikatory dostępu: `public`, `protected`, `private`.
- Inne modyfikatory: `final`, `abstract`, `strictfp`

Modyfikatory dostępu

Choć mamy trzy modyfikatory dostępu (`public`, `protected`, `private`), to w Javie mamy cztery poziomy kontroli (ang. *level or access control*) – czwarty nazywany jest domyślnym (ang. *default access level*) lub pakietowym (ang. *package access level*). W stosunku do klas używamy albo dostępu domyślnego, albo `public`.

Co rozumiemy pod pojęciem: *dostęp do klasy*?

Powiemy, że klasa A ma dostęp do klasy B jeśli klasa A może wykonać jedną z następujących czynności

- Utworzyć instancję klasy B.
- Rozszerzyć klasę B.
- Używać metod i zmiennych klasy B (zależnie od ich modyfikatorów dostępu).

Co rozumiemy pod pojęciem: *dostęp do klasy*?

Powiemy, że klasa A ma dostęp do klasy B jeśli klasa A może wykonać jedną z następujących czynności

- Utworzyć instancję klasy B.
- Rozszerzyć klasę B.
- Używać metod i zmiennych klasy B (zależnie od ich modyfikatorów dostępu).

Co rozumiemy pod pojęciem: *dostęp do klasy*?

Powiemy, że klasa A ma dostęp do klasy B jeśli klasa A może wykonać jedną z następujących czynności

- Utworzyć instancję klasy B.
- Rozszerzyć klasę B.
- Używać metod i zmiennych klasy B (zależnie od ich modyfikatorów dostępu).

Co rozumiemy pod pojęciem: *dostęp do klasy*?

Powiemy, że klasa A ma dostęp do klasy B jeśli klasa A może wykonać jedną z następujących czynności

- Utworzyć instancję klasy B.
- Rozszerzyć klasę B.
- Używać metod i zmiennych klasy B (zależnie od ich modyfikatorów dostępu).

Domyślny poziom kontroli

Listing 1: Plik `Klasa.java`

```
package pakiet1;
class Klasa { }
```

Listing 2: Plik `KlasaLepsza.java`

```
package pakiet2;
import pakiet1.Klasa;
class KlasaLepsza extends Klasa { }
```

Wynik próby uruchomienia

```
Can't access class pakiet1.Klasa. Class or interface must be
public, in same package, or an accessible member class.
import pakiet1.Klasa;
```

Publiczny poziom kontroli

Listing 3: Plik `Klasa.java`

```
package pakiet1;
public class Klasa { }
```

Teraz można napisać

Listing 4: Plik `KlasaLepsza.java`

```
package pakiet2;
import pakiet1.Klasa;
class KlasaLepsza extends Klasa { }
```

final

abstract

strictfp

Czym jest interfejs?

Interfejs jest pewnego rodzaju „umową” określającą co klasa może robić, bez określania w jaki sposób to robi.

Deklarowanie interfejsów

Czym jest interfejs?

O interfejsie można myśleć jak o 100% abstakcyjnej klasie. W konsekwencji metody interfejsu z definicji są publiczne i abstrakcyjne więc nie musimy tego pisać:

```
public abstract void interfaceMethod();
```

zwykle zapisujemy jako

```
void interfaceMethod();
```

Pozostałe cechy interfejsów

- Interfejs deklarujemy za pomocą słowa kluczowego `interface`.
- Wszystkie zmienne interfejsu muszą być publiczne, statyczne i zadeklarowane jako `final` – inymi słowy, w interfejsie można tworzyć tylko stałe. Konsekwencja tego jest taka jak dla metod: ponieważ zmienna muszą być takie, wiec nie trzeba tego pisać.
- Interfejs nie może posiadać modyfikatora `static`.
- Ponieważ metody interfejsu są abstrakcyjne, więc nie mogą być `final`, `strictfp`, `native`.
- Interfejsc może tylko rozszerzać jeden lub więcej interfejsów.
- Interfejs nie może implementować innego interfejsu lub klasy.

Pozostałe cechy interfejsów

- Interfejs deklarujemy za pomocą słowa kluczowego `interface`.
- Wszystkie zmienne interfejsu muszą być publiczne, statyczne i zadeklarowane jako `final` – inymi słowy, w interfejsie można tworzyć tylko stałe. Konsekwencja tego jest taka jak dla metod: ponieważ zmienna muszą być takie, wiec nie trzeba tego pisać.
- Interfejs nie może posiadać modyfikatora `static`.
- Ponieważ metody interfejsu są abstrakcyjne, więc nie mogą być `final`, `strictfp`, `native`.
- Interfejsc może tylko rozszerzać jeden lub więcej interfejsów.
- Interfejs nie może implementować innego interfejsu lub klasy.

Pozostałe cechy interfejsów

- Interfejs deklarujemy za pomocą słowa kluczowego `interface`.
- Wszystkie zmienne interfejsu muszą być publiczne, statyczne i zadeklarowane jako `final` – inymi słowy, w interfejsie można tworzyć tylko stałe. Konsekwencja tego jest taka jak dla metod: ponieważ zmienna muszą być takie, wiec nie trzeba tego pisać.
- Interfejs nie może posiadać modyfikatora `static`.
- Ponieważ metody interfejsu są abstrakcyjne, więc nie mogą być `final`, `strictfp`, `native`.
- Interfejsc może tylko rozszerzać jeden lub więcej interfejsów.
- Interfejs nie może implementować innego interfejsu lub klasy.

Pozostałe cechy interfejsów

- Interfejs deklarujemy za pomocą słowa kluczowego `interface`.
- Wszystkie zmienne interfejsu muszą być publiczne, statyczne i zadeklarowane jako `final` – inymi słowy, w interfejsie można tworzyć tylko stałe. Konsekwencja tego jest taka jak dla metod: ponieważ zmienna muszą być takie, wiec nie trzeba tego pisać.
- Interfejs nie może posiadać modyfikatora `static`.
- Ponieważ metody interfejsu są abstrakcyjne, więc nie mogą być `final`, `strictfp`, `native`.
- Interfejsc może tylko rozszerzać jeden lub więcej interfejsów.
- Interfejs nie może implementować innego interfejsu lub klasy.

Pozostałe cechy interfejsów

- Interfejs deklarujemy za pomocą słowa kluczowego `interface`.
- Wszystkie zmienne interfejsu muszą być publiczne, statyczne i zadeklarowane jako `final` – inymi słowy, w interfejsie można tworzyć tylko stałe. Konsekwencja tego jest taka jak dla metod: ponieważ zmienna muszą być takie, wiec nie trzeba tego pisać.
- Interfejs nie może posiadać modyfikatora `static`.
- Ponieważ metody interfejsu są abstrakcyjne, więc nie mogą być `final`, `strictfp`, `native`.
- Interfejsc może tylko rozszerzać jeden lub więcej interfejsów.
- Interfejs nie może implementować innego interfejsu lub klasy.

Pozostałe cechy interfejsów

- Interfejs deklarujemy za pomocą słowa kluczowego `interface`.
- Wszystkie zmienne interfejsu muszą być publiczne, statyczne i zadeklarowane jako `final` – inymi słowy, w interfejsie można tworzyć tylko stałe. Konsekwencja tego jest taka jak dla metod: ponieważ zmienna muszą być takie, wiec nie trzeba tego pisać.
- Interfejs nie może posiadać modyfikatora `static`.
- Ponieważ metody interfejsu są abstrakcyjne, więc nie mogą być `final`, `strictfp`, `native`.
- Interfejsc może tylko rozszerzać jeden lub więcej interfejsów.
- Interfejs nie może implementować innego interfejsu lub klasy.

Pozostałe cechy interfejsów

- Interfejs deklarujemy za pomocą słowa kluczowego `interface`.
- Wszystkie zmienne interfejsu muszą być publiczne, statyczne i zadeklarowane jako `final` – inymi słowy, w interfejsie można tworzyć tylko stałe. Konsekwencja tego jest taka jak dla metod: ponieważ zmienna muszą być takie, wiec nie trzeba tego pisać.
- Interfejs nie może posiadać modyfikatora `static`.
- Ponieważ metody interfejsu są abstrakcyjne, więc nie mogą być `final`, `strictfp`, `native`.
- Interfejsc może tylko rozszerzać jeden lub więcej interfejsów.
- Interfejs nie może implementować innego interfejsu lub klasy.

Modyfikatory dostępu

Podczas gdy klasa może używać tylko dwóch modyfikatorów dostępu, tj. domyślnego lub `public`, składowe klasy (czyli metody i zmienne) mogą używać wszystkich czterech modyfikatorów, tj. `public`, `protected`, `private` oraz domyślny.

Modyfikator `public`

Dostęp mają wszyscy, bez względu na pakiet w jakim się znajdują (zakładając, że modyfikatory klas na to pozwalają).

Deklarowanie składowych klasy

Modyfikator `public` (odwołania do składowych)

```
package book;
import cert.*;
class Goo {
    public static void main(String[] args) {
        Sludge o = new Sludge();
        o.testIt();
    }
}


package cert;
public class Sludge {
    public void testIt()
    {
        System.out.println("sludge");
    }
}
```

Deklarowanie składowych klasy

Modyfikator `public` (wpływ na dziedziczenie)

```
package cert;
public class Roo {
    public String doRooThings() {
        // imagine the fun code that goes here
        return "fun";
    }
}


package notcert;
import cert.Roo;
class Cloo extends Roo {
    public void testCloo() {
        System.out.println(doRooThings());
    }
}

//cdn
```

Deklarowanie składowych klasy

Modyfikator `public` (wpływ na dziedziczenie)

```
//cd
class Toon {
    public static void main(String[] args) {
        Cloo c = new Cloo();
        System.out.println(c.doRooThings());
        //No problem; method is public
    }
}
```

Modyfikator `private`

Składowe prywatne są przeciwieństwem składowych publicznych – dosęp do nich ma tylko klasa, w której zostały zadeklarowane.

Modyfikator `private` (odwołania do składowych 1/2)

```
package cert;
public class Roo {
    private String doRooThings() {
        // imagine the fun code that goes here,
        // but only the Roo class knows
        return "fun";
    }
}

package notcert;
import cert.Roo;
class UseARoo {
    public void testIt() {
        Roo r = new Roo(); // So far so good;
                           // class Roo is public
        System.out.println(r.doRooThings()); //Compiler error!
    }
}
```

Modyfikator `private` (odwołania do składowych 2/2)

```
cannot find symbol
symbol : method doRooThings()
```

Modyfikator `private` (wpływ na dziedziczenie 1/2)

```
package cert;
public class Roo {
    private String doRooThings() {
        // imagine the fun code that goes here,
        // but no other classwill know
        return "fun";
    }
}


package cert;
//Cloo and Roo are in the same package
class Cloo extends Roo {//Still OK, superclass Roo is public
    public void testCloo() {
        System.out.println(doRooThings()); //Compiler error!
    }
}
```

Modyfikator `private` (wpływ na dziedziczenie 2/2)

```
%javac Cloo.java
Cloo.java:4: Undefined method: doRooThings()
System.out.println(doRooThings());
1 error
```

Modyfikator `protected`

W przypadku modyfikatora `protected` do składowej mają dostęp wszystkie klasy potomne (nawet te z innych pakietów) oraz klasy z danego pakietu.

Modyfikator `protected` (odwołania do składowych 1/2)

```
package certification;
public class OtherClass {
    void testIt() {
        // No modifier means method has default
        // access
        System.out.println("OtherClass");
    }
}

package somethingElse;
import certification.OtherClass;
class AccessClass {
    static public void main(String[] args) {
        OtherClass o = new OtherClass();
        o.testIt();
    }
}
```

Modyfikator `protected` (odwołania do składowych 2/2)

```
No method matching testIt() found in class
certification.OtherClass.
o.testIt();
```

Deklarowanie składowych klasy

Modyfikator `protected` (odwołania do składowych)

```
package certification;
public class Parent {
    protected int x = 9; // protected access
}


package other; // Different package
import certification.Parent;
class Child extends Parent {
    public void testIt() {
        System.out.println("x is " + x);
        // No problem; Child
        // inherits x
    }
}
```

Modyfikator `protected` (odwołania do składowych 1/2)

```
package certification;
public class Parent {
    protected int x = 9; // protected access
}

package other;
import certification.Parent;
class Child extends Parent {
    public void testIt() {
        System.out.println("x is " + x); // No problem;
                                         // Childinherits x
        Parent p = new Parent(); // Can we access x using the
                                 // p reference?
        System.out.println("X in parent is " + p.x);
        // Compiler error!
    }
}
```

Modyfikator `protected` (odwołania do składowych 2/2)

```
%javac -d . other/Child.java
other/Child.java:9: x has protected access in certification.Par-
ent
System.out.println("X in parent is " + p.x);
                                          ^
1 error
```

Modyfikator domyślny

W przypadku domyślnego poziomu dostępu do składowej mają dostęp klasy z tego samego pakietu.

Modyfikator `protected` (odwołania do składowych 1/2)

```
package certification;
public class OtherClass {
    void testIt() {
        // No modifier means method has default
        // access
        System.out.println("OtherClass");
    }
}

package other; // Different package
import certification.Parent;
class Child extends Parent {
    public void testIt() {
        System.out.println("x is " + x);
    }
}
```

Modyfikator `protected` (odwołania do składowych 2/2)

```
Child.java:4: Undefined variable: x
System.out.println("Variable x is " + x);
1 error
```

Deklarowanie składowych klasy

Modyfikator `protected` (odwołania do składowych)

```
package certification;
public class Parent{
    int x = 9; // default access
}


package certification;
class Child extends Parent{
    static public void main(String[] args) {
        Child sc = new Child();
        sc.testIt();
    }

    public void testIt() {
        System.out.println("Variable x is " + x); // No
            problem;
    }
}
```

Deklarowanie zmiennych lokalnych

Modyfikator `protected` (odwołania do składowych)

Poniższy zapis nie jest poprawny

```
class Foo {
   void doStuff() {
      private int x = 7;
         this.doMore(x);
      }
}
```

Jedyny modyfikator jaki można zastosować do zmiennych lokalnych to `final`.

Modyfikatory dostępu – podsumowanie

Widoczność	`public`	`protected`	`private`	domyślny
Ta sama klasa				
Inna klasa z tego samego pakietu				
Podklasa z tego samego pakietu				
Podklasa z innego pakietu				
Klasa z innego pakietu				

Inne modyfikatory dostępu

`final`, `abstract`, `synchronized`, `native`, `strictfp`

Modyfikator `final`

Zabezpiecza metodę przed jej przesłonięciem w klasie potomnej (ang. *override*)

Deklarowanie składowych klasy

Modyfikator `final` (w przypadku metody)

```
class SuperClass{
    public final void showSample() {
        System.out.println("One thing.");
    }
}

class SubClass extends SuperClass{
    public void showSample() {
        // Try to override the final
        // superclass method
        System.out.println("Another thing.");
    }
}
```

Modyfikator `final` (w przypadku metody)

```
%javac FinalTest.java
FinalTest.java:5: The method void showSample() declared in class
SubClass cannot override the final method of the same signature
declared in class SuperClass.
Final methods cannot be overridden.
public void showSample() { }
1 error
```

Deklarowanie składowych klasy

Modyfikator `final` (w przypadku argumentu)

```
public Record getRecord(int fileNumber, int recNumber) {}

public Record getRecord(int fileNumber, final int
    recordNumber) {}
```

Modyfikator `abstract`

Metoda abstrakcyjna to metoda, która została zadeklarowana, ale nie została zaimplementowana.

Deklarowanie składowych klasy

Modyfikator `abstract` (w przypadku metody)

```
public class IllegalClass{
    //illegal if class is not abstract
    public abstract void doIt();
}
```

```
IllegalClass.java:1: class IllegalClass must be declared
abstract.
It does not define void doIt() from class IllegalClass.
public class IllegalClass{
1 error
```

Deklarowanie składowych klasy

Modyfikator `abstract` (w przypadku metody)

O ile nie może być niebastrakcyjnej klasy z metodami abstrakcyjymi to może być abstrakcyjna klasa z (tylko) metodami nieabstrakcyjnymi.

```
public abstract class LegalClass{
    void goodMethod() {
        // lots of real implementation code here
    }
}
```

Deklarowanie składowych klasy

Modyfikator `abstract` (w przypadku metody)

Pytanie: ile metod jest w klasie `Mini`

```
public abstract class Vehicle {
    private String type;
    public abstract void goUpHill();// Abstract method
    public String getType() {       // Non-abstract method
        return type;
    }
}

public abstract class Car extends Vehicle {
    public abstract void goUpHill(); // Still abstract
    public void doCarThings() {
        // special car code goes here
    }
}

public class Mini extends Car {
    public void goUpHill() {
        // Mini-specific going uphill code
    }
}
```

Modyfikator `abstract` (w przypadku metody)

Pytanie: czy ten kod się skompiluje?

```
public abstract class A {
    abstract void foo();
}

class B extends A {
    void foo(int l) { }
}
```

To jest przykład na przeładowanie (przeciążenie) (ang. *overload*). Klasa B nie zawiera w sobie implementacji metody `foo`.

Modyfikator `abstract` (w przypadku metody)

Pytanie: czy ten kod się skompiluje?

```
public abstract class A {
    abstract void foo();
}

class B extends A {
    void foo(int I) { }
}
```

To jest przykład na przeładowanie (przeciążenie) (ang. *overload*). Klasa B nie zawiera w sobie implementacji metody `foo`.

Deklarowanie składowych klasy

Modyfikator `abstract` (w przypadku metody)

`abstract` + `static` = illegal

```
abstract static void doStuff();
```

```
MyClass.java:2: illegal combination of modifiers: abstract and
static
abstract static void doStuff();
```

Deklarowanie składowych klasy

Modyfikator `synchronized`

`synchronized` oznacza, że metoda nie może być wywołana przez więcej niż jeden wątek w tym samym czasie.

```
public synchronized Record retrieveUserInfo(int id) { }
```

Modyfikator ten dotyczy tylko metod (nie zmiennych, nie klas).

Modyfikator `native`, `strictfp`

Modyfikator `native` oznacza, że metoda (i tylko metoda) zaimplementowana została w sposób zależny platformowo, np. w języku C. Ciało takiej metody zastępujemy znakiem średnika (podobnie jak ciało metody abstrakcyjnej).

Deklarowanie składowych klasy

Deklarowanie metod ze zmienną listą argumentów

```
Legal:
void doStuff(int... x) { } // expects from 0 to many ints
                           // as parameters
void doStuff2(char c, int... x) { } // expects first a char,
                                    // then 0 to many ints
void doStuff3(Animal... animal) { } // 0 to many Animals

Illegal:
void doStuff4(int x...) { } // bad syntax
void doStuff5(int... x, char... y) { } // too many var-args
void doStuff6(String... s, byte b) { } // var-arg must be
    last
```

Deklarowanie składowych klasy

Konstruktor

```
class Foo {
protected Foo() { } // this is Foo's constructor

protected void Foo() { } // this is a badly named,
                         // but legal, method
}
```

Konstruktor

Konstruktor nie może być poprzedzony modyfikatorem `static`, `final` ani `abstract`.

Deklarowanie składowych klasy

Konstruktor

```
class Foo {
protected Foo() { } // this is Foo's constructor

protected void Foo() { } // this is a badly named,
                         // but legal, method
}
```

Konstruktor

Konstruktor nie może być poprzedzony modyfikatorem `static`, `final` ani `abstract`.

Deklarowanie składowych klasy

Konstruktor

```
class Foo2 {
    // legal constructors
    Foo2() { }
    private Foo2(byte b) { }
    Foo2(int x) { }
    Foo2(int x, int... y) { }
}
```

Deklarowanie składowych klasy

Konstruktor

```
class Foo2 {
   // illegal constructors
   void Foo2() { }//it's a method, not a constructor
   Foo() { }//not a method or a constructor
   Foo2(short s);//looks like an abstract method
   static Foo2(float f) { }//can't be static
   final Foo2(long x) { }//can't be final
   abstract Foo2(char c) { }//can't be abstract
   Foo2(int... x, int t) { }//bad var-arg syntax
}
```

Deklarowanie składowych klasy

Zmienne

W Javie mamy dwa rodzaje zmiennych

- **Zmienne proste** `char`, `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`

```
byte b;
boolean myBooleanPrimitive;
int x, y, z;
```

- **Zmienne referencyjne** czyli zmienne używane w stosunku do obiektów. Zmienna referencyjna pozwala na uzyskanie dostępu do obiektów typu takiego jak zmienna referencyjna lub będących podtypem zaeklarowanego typu.

```
Object o;
Dog myNewDogReferenceVariable;
String s1, s2, s3;
```

Deklarowanie składowych klasy

Zmienne instancyjne

```
class Employee {
    // define fields (instance variables) for employee
        instances
    private String name;
    private String title,
    private String manager;
    // other code goes here including access methods for
        private
    // fields
}
```

Deklarowanie składowych klasy

Zmienne lokalne

```
class TestServer {
    public void logIn() {
        int count = 10;
    }
}
```

Deklarowanie składowych klasy

Zmienne lokalne

```
class TestServer {
    public void logIn() {
        int count = 10;
    }

    public void doSomething(int i) {
        count = i; // Won't compile! Can't access count
            outside
                    // method logIn()
        }
}
```

Deklarowanie składowych klasy

Zmienne instancyjne i lokalne (shadowing)

```
class TestServer {
    int count = 9;//Declare an instance variable named count
    public void logIn() {
        int count = 10;//Declare a local variable named count
        System.out.println("local variable count is "+count);
    }

    public void count() {
        System.out.println("instance variable count is " +
            count);
    }

    public static void main(String[] args) {
        new TestServer().logIn();
        new TestServer().count();
    }
}
```

Deklarowanie składowych klasy

Zmienne instancyjne i lokalne (shadwing)

```
illegal
class Foo {
    int size = 27;
    public void setSize(int size) {
        size = size; // ??? which size equals which size???
    }
}

legal:
class Foo {
    int size = 27;
    public void setSize(int size) {
        this.size = size; // this.size means the current
            object's
                          // instance variable, size. The size
                          // on the right is the parameter
    }
}
```

Deklarowanie składowych klasy

Tablice

```
Declaring an Array of Primitives
int[] key;   // Recommended
int key []; // Legal but less readable

Declaring an Array of Object References
Thread[] threads; // Recommended
Thread threads []; // Legal but less readable

String[][][] occupantName;
String[] managerName [];//Legal but not recommended
```

Statyczne zmienne i metody

Modyfikator `static` wykorzysywany jest do utworzenia zmiennych i metod stniejących niezależnie od jakichkolwiek obiektów danej klasy.

Deklarowanie składowych klasy

Typ wyliczeniowy (enum)

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING };

CoffeeSize cs = CoffeeSize.BIG;
```

Deklarowanie składowych klasy

Typ wyliczeniowy (enum) na zewnątrz klasy

```
// this cannot be private or protected
enum CoffeeSize { BIG, HUGE, OVERWHELMING }

class Coffee {
    CoffeeSize size;
    }

public class CoffeeTest1 {
    public static void main(String[] args) {
        Coffee drink = new Coffee();
        drink.size = CoffeeSize.BIG;
        // enum outside class
    }
}
```

Deklarowanie składowych klasy

Typ wyliczeniowy (enum) wewnątrz klasy

```
class Coffee2 {
    enum CoffeeSize {BIG, HUGE, OVERWHELMING }
    CoffeeSize size;
}

public class CoffeeTest2 {
    public static void main(String[] args) {
        Coffee2 drink = new Coffee2();
        drink.size = Coffee2.CoffeeSize.BIG;
        // enclosing class
        // name required
    }
}
```

Deklarowanie składowych klasy

Typ wyliczeniowy (enum) nie może być deklarowany wewnątrz metody

```
public class CoffeeTest1 {
    public static void main(String[] args) {
        enum CoffeeSize { BIG, HUGE, OVERWHELMING }
        Coffee drink = new Coffee();
        drink.size = CoffeeSize.BIG;
    }
}
```

Opcjonaln średnik

```
//semicolon at the end of the enum declaration is optional
enum CoffeeSize { BIG, HUGE, OVERWHELMING };
```