Programowanie i projektowanie obiektowe

Java

Paweł Daniluk

Wydział Fizyki

Jesień 2013



Przegląd składni

Komentarze

```
/* This is a multi-line comment.
   It may occupy more than one line. */
```

```
// This is an end-of-line comment
```

Zmienne i przypisania

```
int myInt;                  /* Declaring an uninitialized variable
                               called 'myInt', of type 'int' */
myInt = 35;                 // Initializing the variable
int myOtherInt = 35;        /* Declaring and initializing the
                               variable at the same time */
```

Przegląd składni c.d.

Instrukcja warunkowa

```
if (i == 3) doSomething();
```

```
if (i == 2)
    doSomething();
else
    doSomethingElse();
```

```
if (i == 3) {
    doSomething();
} else if (i == 2) {
    doSomethingElse();
} else {
    doSomethingDifferent();
}
```

Przegląd składni c.d.

Instrukcja wyboru

```
switch (ch) {
    case 'A':
        doSomething(); // Triggered if ch == 'A'
        break;

    case 'B':
    case 'C':
        doSomethingElse(); // Triggered if ch == 'B'
        break;             // or ch == 'C'

    default:
        doSomethingDifferent(); // Triggered in any other case
        break;
}
```

Przegląd składni c.d.

Pętla while

```
while (i < 10) {
    doSomething();
}
```

```
// doSomething() is called at least once
do {
    doSomething();
} while (i<10);
```

Przegląd składni c.d.

Pętla for

```
for (int i = 0; i < 10; i++) {
    doSomething();
}
```

```
// A more complex loop using two variables
for (int i = 0, j = 9; i < 10; i++, j -= 3) {
    doSomething();
}
```

```
for (;;) {
    doSomething();
}
```

Tablice

W Javie podstawowym "odpowiednikiem" Pythonowych list są tablice (ang. *arrays*), które mają z góry ustaloną długość.

```
int[] numbers = new int[5];
numbers[0] = 2;
numbers[1] = 5;
int x = numbers[0];
```

Inicjalizacja

```
// Long syntax
int[] numbers = new int[5] {20, 1, 42, 15, 34};
// Short syntax
int[] numbers2 = {20, 1, 42, 15, 34};
```

Tablice c.d.

Tablice wielowymiarowe

```
int[][] numbers = new int[3][3];
number[1][2] = 2;

int[][] numbers2 = {{2, 3, 2}, {1, 2, 6}, {2, 4, 5}};
```

Wiersze mogą być różnej długości

```
// Initialization of the first dimension only
int[][] numbers = new int[2][];

numbers[0] = new int[3];
numbers[1] = new int[2];
```

Klasy w Javie

```
class Pusta {
}
```

Klasy w Javie

```
class Pusta {
}
```

Atrybuty

```
class Osoba {
    String imie;
    String nazwisko;
}
```

Klasy w Javie

```
class Pusta {
}
```

Atrybuty

```
class Osoba {
    String imie;
    String nazwisko;
}
```

Metody

```
class Osoba {
    String imie;
    String nazwisko;
    String imie() { return imie; }
    String nazwisko() { return nazwisko; }
}
```

Dziedziczenie

```
class Animal {
    String talk() { return "?!?!?"; }
}

class Cat extends Animal {
    String talk() { return "Meow!"; }
}

class Dog extends Animal {
    String talk() { return "Woof!"; }
}
```

Dziedziczenie c.d.

```
static void main() {
    Animal a=new Cat();
    Dog d=new Dog();

    System.out.println(a.talk());

    a=d;                          // Takie przypisanie jest ok.

    Cat c=a;                      // A takie nie.

    if(a instanceof Cat) {        // Za to wolno tak.
        Cat c = (Cat) a;
    }
}
```

Dziedziczenie c.d.

Przysłanianie metod

Jeżeli w podklasie jest zdefiniowana metoda o takiej samej nazwie jak w nadklasie, to dla każdego obiektu podklasy będzie ona wykonywana, niezależnie od typu referencji, która wskazuje na obiekt.

Anotacja oznaczająca metodę przysłaniającą:
`@Override`

Odwoływanie się do elementów nadklasy

`super` – oznacza referencję do nadklasy
`this` – oznacza referencję do samej siebie

Przykład

```
class KolorowyKlocek extends Klocek {
    String kolor;

    public String toString() {
        return super.toString()+" koloru "+kolor;
    }

    setKolor(String kolor) {
        this.kolor=kolor;
    }
}
```

Konstruktory

Konstruktor

```
class Osoba {
    String imie;
    String nazwisko;
    int wiek;

    Osoba(String imie, String nazwisko) {
        this.imie = imie;
        this.nazwisko = nazwisko;
    }

    Osoba(String imie, String nazwisko, int wiek) {
        this(imie, nazwisko);
        this.wiek = wiek;
    }
}
```

Wywołanie innego konstruktora musi być pierwszą instrukcją.

Konstruktory c.d.

Konstruktor, a dziedziczenie

Definiując konstruktor podklasy można posłużyć się konstruktorem nadklasy.

Konstruktor, a dziedziczenie

```
class Student extends Osoba {
    int nrIndeksu;

    Student(String imie, String nazwisko, int nrIndeksu) {
        super(imie, nazwisko);
        this.nrIndeksu = nrIndeksu;
    }
}
```

Przeciążanie

Przeciążanie metod

W klasie mogą być zdefiniowanych wiele metod o tej samej nazwie różniących się liczbą i typem argumentów.

```
class Kalkulator {

    int dodaj(int a, int b) { ... }

    double dodaj(double a, double b) { ... }
}
```

Typy danych (w Javie)

Typy pierwotne

- typ wartości logicznych: `boolean`
- typy całkowitoliczbowe: `byte`, `short`, `int`, `long`, `char`
- typy zmiennopozycyjne: `float`, `double`

Typy danych (w Javie)

Typy pierwotne

- typ wartości logicznych: `boolean`
- typy całkowitoliczbowe: `byte`, `short`, `int`, `long`, `char`
- typy zmiennopozycyjne: `float`, `double`

Typy referencyjne

- typy klas
- typy interfejsów
- typy tablic

Typy pierwotne

Zmienna typu pierwotnego zawiera pojedynczą wartość.

```
int i;
double f=0.5;
```

typ	wartości
`boolean`	`true`, `false`
`byte`	$-128 \div 127$
`short`	$-32,768 \div 32,767$
`int`	$-2,147,483,648 \div 2,147,483,647$
`long`	$-9,223,372,036,854,775,808 \div 9,223,372,036,854,775,807$
`char`	$0 \div 255$
`float`	
`double`	

Typy referencyjne

Zmienna typu referencyjnego ma wartość `null` lub wskazuje na wartość odpowiedniego typu. Wartości typów referencyjnych mogą się zmieniać w czasie.

```
int i,j;

i = 5;
j = i;
System.out.format("i: %d j: %d\n", i, j);

j = 3;
System.out.format("i: %d j: %d\n", i, j);
```

```
i: 5 j: 5
i: 5 j: 3
```

Typy referencyjne c.d.

```
class Test {
        int val;
}

Test obI = new Test();
obI.val = 5;
Test obJ = obI;
System.out.format("obI.val: %d obJ.val: %d\n", obI.val, obJ.val)

obJ.val = 3;
System.out.format("obI.val: %d obJ.val: %d\n", obI.val, obJ.val)
```

```
obI.val: 5 obJ.val: 5
obI.val: 3 obJ.val: 3
```

Organizacja kodu w Javie

Klasy i interfejsy

Każda klasa (lub interfejs) umieszczana jest w osobnym pliku (z rozszerzeniem .java).

Pakiety

Pliki z klasami mogą być umieszczane w drzewiastej strukturze analogicznej do katalogów w systemie plików.

Modyfikatory dostępu

Dostępność elementów

Modyfikator	Wewnątrz klasy	W innej klasie w tym samym pakiecie	Podklasa w innym pakiecie	Dowolna klasa w innym pakiecie
`private`	tak	nie	nie	nie
domyślnie	tak	tak	nie	nie
`protected`	tak	tak	tak	nie
`public`	tak	tak	tak	tak

Kapsułkowanie

Często opłaca się deklarować atrybuty z modyfikatorem `private` i udostępniać metody do pobierania i zmieniania ich wartości.

Przykład

```
public class A {
    private int val;
    private boolean cond;

    int getVal() {
        return val;
    }

    void setVal(int val) {
        this.val = val;
    }

    boolean isCond() {
        return cond;
    }
}
```

Modyfikatory

Modyfikatory klas

`abstract` – Klasa służy wyłącznie jako węzeł w hierarchii klas. Nie można tworzyć obiektów należących do niej.
`final` – Nie można dziedziczyć z klas oznaczonych tym atrybutem.
`static` –

Modyfikatory metod

`abstract` – Stosowany z klasach abstrakcyjnych. Oznacza, że metoda o takiej nazwie i argumentach musi być zdefiniowana we wszystkich podklasach.
`final` – Nie można przesłaniać metod oznaczonych tym atrybutem.
`static` –

Modyfikator `static`

`static` oznacza, że element klasy nie należy do żadnej jej instancji.

```
class Foo {
    static int bar;
    float baz;
}

Foo f=new Foo();

//Poprawnie
Foo.bar=10
f.baz=Foo.bar;

//Niepoprawnie
Foo.baz;
f.bar;
```

Modyfikator `static` c.d.

Statyczne metody nie mają dostępu do atrybutów instancji.

```
class Foo {
    static int bar;
    float baz;

    void qux() { //Dobrze
        int i=bar;
        baz=2*baz;
    }

    static void bag() {
        baz+=3.14;     // Zle
        int j=2*bar; // Dobrze
    }
}
```

Klasy zagnieżdżone

Zwykłe klasy (niezagnieżdżone)

```
class Foo {
// Class members
}
```

Klasy zagnieżdżone

```
class Foo { // Top-level class
    class Bar { // Nested class
    }
}
```

Klasy zagnieżdżone c.d.

Klasy lokalne

```
class Foo {
    void bar() {
        class Foobar {  // Local class within a method
        }
    }
}
```

Klasy anonimowe

```
class Foo {
    void bar() {
        new Object() {
            // Creation of a new anonymous class extending Objec
        };
    }
}
```

Próba podsumowania

System typów

- W Javie typy mają zarówno zmienne, jak i obiekty. Nie można przypisać referencji do obiektu na zmienną, której tym nie zawiera typu obiektu.
- To wymusza tworzenie nadklas. (-)
- Typowanie zmiennych umożliwia przeciążanie. (+)
- W przypadku dużych kłopotów trzeba stosować rzutowanie (-)

Kapsułkowanie i prawa dostępu

- Można zabezpieczać klasy i ich komponenty (+)
- Kapsułkowanie jest koniecznością (-)

Próba podsumowania c.d.

Funkcje i klasy

- W Javie funkcje nie są wartościami (nie ma *first-class functions*) (-)
- Wcale nie ma funkcji (-)
- Trzeba tworzyć klasy z metodami statycznymi (-)
- Żeby przekazać funkcję trzeba stworzyć klasę z odpowiednią metodą i przekazać jej instancję. (-)
- Klasy anonimowe pomagają. (+)

Wielodziedziczenie

- Brak (-)

Interfejsy

Interfejs to szczególny przypadek klasy abstrakcyjnej, która nie implementuje żadnych metod.

Pozwalają na wielodziedziczenie w wersji dla ubogich.

Przykład

```
class Pracownik extends Osoba
        // dziedziczenie po klasie
class Samochod implements Pojazd, Towar
        // dziedziczenie po kilku interfesjach
class Chomik extends Ssak implements Puchate, DoGlaskania
        // dziedziczenie po klasie i kilku interfejsach
```

Interfejsy c.d.

Przykład

```
interface ActionListener {
      void actionSelected(int action);
}

interface RequestListener {
    int requestReceived();
}

class ActionHandler implements ActionListener, RequestListener {
    void actionSelected(int action) {}

    public int requestReceived() {}
}

//Calling method defined by interface
RequestListener listener = new ActionHandler();
    /*ActionHandler can represented be as RequestListener...*/
listener.requestReceived();
    /*...and thus is known to implement requestReceived() method*/
```

Typy generyczne

Dzięki polimorfizmowi możemy mieć kontenery zawierające dowolne obiekty, ale żeby zrobić coś z ich zawartością konieczne jest rzutowanie.

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0);          // Run time error
```

Gdybyśmy umieli powiedzieć, że v będzie przechowywać wyłącznie ciągi znaków, wykrylibyśmy problem już podczas kompilacji.

```
List<String> v = new ArrayList<String>();
v.add("test");
Integer i = v.get(0); // (type error)  Compile time error
```

Typy generyczne c.d.

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Klasa generyczna

Definicja

```
/* This class has two type variables, T and V. T must be
   a subtype of ArrayList and implement Formattable interface */

public class Mapper<T extends ArrayList & Formattable, V> {
    public void add(T array, V item) {
        // array has add method because it is an ArrayList subclass
        array.add(item);
    }
}
```

Zastosowanie

```
/* Mapper is created for CustomList as T and Integer as V.
   CustomList must be a subclass of ArrayList and
   implement Formattable */

Mapper<CustomList, Integer> mapper =
    new Mapper<CustomList, Integer>();
```

Klasa generyczna c.d.

```
/* Any Mapper instance with CustomList as the first parameter
may be used regardless of the second one.*/

Mapper<CustomList, ?> mapper;
mapper = new Mapper<CustomList, Boolean>();
mapper = new Mapper<CustomList, Integer>();
```

```
/* Will not accept types that use anything but
a subclass of Number as the second parameter */

void addMapper(Mapper<?, ? extends Number> mapper) {
}
```

Generyczne metody

```
class Mapper {
    // The class itself is not generic, the constructor is
    <T, V> Mapper(T array, V item) {
    }
}

/* This method will accept only arrays of the same type as
the searched item type or its subtype*/
static <T, V extends T> boolean contains(T item, V[] arr) {
    for (T currentItem : arr) {
        if (item.equals(currentItem)) {
            return true;
        }
    }
    return false;
}
```

Generyczne intefejsy

```
interface Expandable<T extends Number> {
    void addItem(T item);
}

// This class is parametrized
class Array<T extends Number> implements Expandable<T> {
    void addItem(T item) {
    }
}

// And this is not and uses an explicit type instead
class IntegerArray implements Expandable<Integer> {
    void addItem(Integer item) {
    }
}
```

Kolejna próba podsumowania

Konieczność stosowania interfejsów i typów generycznych wynika z braku wielodziedziczenia i statycznego typowania. Przy typowaniu dynamicznym (*duck typing*) wystąpienie obiektu niewłaściwej klasy (lub brak atrybutu albo metody) może zostać wykryte dopiero podczas pracy programu. W Javie jest to wykrywane na etapie kompilacji (o ile nie stosuje się rzutowania).

Typy generyczne pozwalają uniknąć rzutowania. Kosztem jest skomplikowany kod.

Java™ Platform, Standard Edition 6

Pakiety

java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework

Java™ Platform, Standard Edition 6

Pakiety

java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing beans – components based on the JavaBeansTM architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.annotation	Provides library support for the Java programming language annotation facility.
java.lang.instrument	Provides services that allow Java programming language agents to instrument programs running on the JVM.
java.lang.management	Provides the management interface for monitoring and

Java™ Platform, Standard Edition 6

Pakiety

java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing beans – components based on the JavaBeansTM architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.annotation	Provides library support for the Java programming language annotation facility.
java.lang.instrument	Provides services that allow Java programming language agents to instrument programs running on the JVM.
java.lang.management	Provides the management interface for monitoring and management of the Java virtual machine as well as the operating system on which the Java virtual machine is running.
java.lang.ref	Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
java.lang.reflect	Provides classes and interfaces for obtaining reflective information about classes and objects.
java.math	Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
java.net	Provides the classes for implementing networking applications.
java.nio	Defines buffers, which are containers for data, and provides an overview of the other NIO packages.
java.nio.channels	Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations.
java.nio.channels.spi	Service-provider classes for the java.nio.channels package.
java.nio.charset	Defines charsets, decoders, and encoders, for translating between bytes and Unicode characters.
java.nio.charset.spi	Service-provider classes for the java.nio.charset package.
java.rmi	Provides the RMI package.
java.rmi.activation	Provides support for RMI Object Activation.
java.rmi.dgc	Provides classes and interface for RMI distributed garbage-collection (DGC).
java.rmi.registry	Provides a class and two interfaces for the RMI registry.
java.rmi.server	Provides classes and interfaces for supporting the server side of RMI.
java.security	Provides the classes and interfaces for the security framework.
java.security.acl	The classes and interfaces in this package have been superseded by classes in the java.security package.
java.security.cert	Provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths.
java.security.interfaces	Provides interfaces for generating RSA (Rivest, Shamir and Adleman AsymmetricCipher algorithm) keys as defined in the RSA Laboratory Technical Note PKCS#1

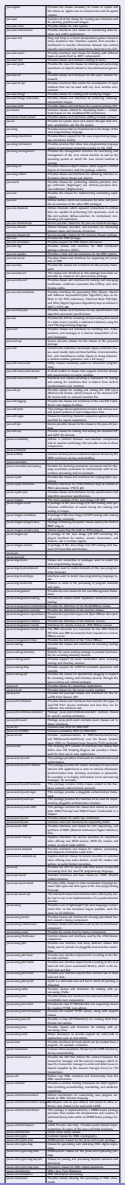
Java™ Platform, Standard Edition 6

Pakiety

java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing beans – components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.annotation	Provides library support for the Java programming language annotation facility.
java.lang.instrument	Provides services that allow Java programming language agents to instrument programs running on the JVM.
java.lang.management	Provides the management interface for monitoring and management of the Java virtual machine as well as the operating system on which the Java virtual machine is running.
java.lang.ref	Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
java.lang.reflect	Provides classes and interfaces for obtaining reflective information about classes and objects.
java.math	Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
java.net	Provides the classes for implementing networking applications.
java.nio	Defines buffers, which are containers for data, and provides an overview of the other NIO packages.
java.nio.channels	Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations.
java.nio.channels.spi	Service-provider classes for the java.nio.channels package.
java.nio.charset	Defines charsets, decoders, and encoders, for translating between bytes and Unicode characters.
java.nio.charset.spi	Service-provider classes for the java.nio.charset package.
java.rmi	Provides the RMI package.
java.rmi.activation	Provides support for RMI Object Activation.
java.rmi.dgc	Provides classes and interface for RMI distributed garbage-collection (DGC).
java.rmi.registry	Provides a class and two interfaces for the RMI registry.
java.rmi.server	Provides classes and interfaces for supporting the server side of RMI.
java.security	Provides the classes and interfaces for the security framework.
java.security.acl	The classes and interfaces in this package have been superseded by classes in the java.security package.
java.security.cert	Provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths.
java.security.interfaces	Provides interfaces for generating RSA (Rivest, Shamir and Adleman AsymmetricCipher algorithm) keys as defined in the RSA Laboratory Technical Note PKCS@1, and DSA (Digital Signature Algorithm) keys as defined in NIST's FIPS-186.
java.security.spec	Provides classes and interfaces for key specifications and algorithm parameter specifications.
java.sql	Provides the API for accessing and processing data stored in a data source (usually a relational database) using the JavaTM programming language.
java.text	Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.
java.text.spi	Service provider classes for the classes in the java.text package.
java.util	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).
java.util.concurrent	Utility classes commonly useful in concurrent programming.
java.util.concurrent.atomic	A small toolkit of classes that support lock-free thread-safe programming on single variables.
java.util.concurrent.locks	Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors.
java.util.jar	Provides classes for reading and writing the JAR (Java ARchive) file format, which is based on the standard ZIP file format with an optional manifest file.
java.util.logging	Provides the classes and interfaces of the JavaTM 2 platform's core logging facilities.
java.util.prefs	This package allows applications to store and retrieve user and system preference and configuration data.
java.util.regex	Classes for matching character sequences against patterns specified by regular expressions.
java.util.spi	Service provider classes for the classes in the java.util packages.
java.util.zip	Provides classes for reading and writing the standard ZIP and GZIP file formats.
javax.accessibility	Defines a contract between user-interface components and an assistive technology that provides access to those components.
javax.activation	
javax.activity	Contains Activity service related exceptions thrown by the ORB machinery during unmarshalling.
javax.annotation	
javax.annotation.processing	Facilities for declaring annotation processors and for allowing annotation processors to communicate with an annotation processing tool environment.
javax.crypto	Provides the classes and interfaces for cryptographic operations.
javax.crypto.interfaces	Provides interfaces for Diffie-Hellman keys as defined in RSA Laboratories' PKCS @3.
javax.crypto.spec	Provides classes and interfaces for key specifications and algorithm parameter specifications.
javax.imageio	The main package of the Java Image I/O API.
javax.imageio.event	A package of the Java Image I/O API dealing with synchronous notification of events during the reading and writing of images.
javax.imageio.metadata	A package of the Java Image I/O API dealing with reading and writing metadata.
javax.imageio.plugins.bmp	Package containing the public classes used by the built-in BMP plug-in.
javax.imageio.plugins.jpeg	Classes supporting the built-in JPEG plug-in.
javax.imageio.spi	A package of the Java Image I/O API containing the plug-in interfaces for readers, writers, transcoders, and streams, and a runtime registry.
javax.imageio.stream	A package of the Java Image I/O API dealing with low-level I/O from files and streams.
javax.jws	
javax.jws.soap	
javax.lang.model	Classes and hierarchies of packages used to model the Java programming language.
javax.lang.model.element	Interfaces used to model elements of the Java programming language.
javax.lang.model.type	Interfaces used to model Java programming language types.
javax.lang.model.util	Utilities to assist in the processing of program elements and types.
javax.management	Provides the core classes for the Java Management Exten-

Java^{TM} Platform, Standard Edition 6

Pakiety



Pakiety



Wzorce projektowe

Sformułowanie wyważone

Użyteczność wzorców projektowych zależy od języka programowania.

Sformułowanie radykalne

Wzorce projektowe służą do maskowania niedoskonałości języka.

Wzorce projektowe c.d.

Singleton

W Pythonie można zastosować moduł. Atrybuty, których nie ma nie kosztują.

Strategy

W Pythonie można posługiwać się funkcjami.

Factory

W Pythonie można zdefiniować metode `__new__`.

Klasy i dziedziczenie

Przynależność do klasy może oznaczać:

- przynależność do zbioru,
- posiadanie konkretnych odpowiedzialności.

W Javie

Dla każdego rodzaju parametrów metody musi istnieć klasa (lub interfejs), który go opisuje. Występuje konieczność definiowania klas czysto abstrakcyjnych (ang. *pure abstract class*).

W Pythonie

Klasy definiuje się wtedy, gdy można zdefiniować ich metody.

Może występować konieczność definiowania owijaczy (ang. *wrappers*), aby ukryć typ obiektu owijanego.

Dziedziczenie vs. składanie

Klasę posiadającą funkcjonalność (lub część funkcjonalności) innej można realizować przy pomocy:

- dziedziczenia
- składania

Dziedziczenie

- dziedziczone jest wszystko (nie należy ograniczać)
- tylko jedna nadklasa (albo wielodziedziczenie)
- podstawialność (Czy w każdym miejscu, gdzie akceptowana jest instancja nadklasy, można podać instancję podklasy?)

Składanie

- elementy składowej trzeba "ręcznie" eksponować
- dowolnie wiele składowych
- brak podstawialności (bo nie ma dziedziczenia)

Formalna poprawność programów

Dzięki typowaniu statycznemu na etapie kompilacji można wykryć szereg błędów:

- odwołanie do nieistniejącej metody (lub atrybutu)
- literówki
- błędy logiczne polegające na użyciu obiektu niewłaściwego typu
- niewłaściwą liczbę argumentów funkcji/metody

Względy praktyczne

Nie ma jednego najlepszego języka programowania.

Kryteria

- swoboda projektowania vs. możliwość formalnej weryfikacji poprawności
- rozwiązania dostosowane do problemu vs. standardowe wzorce/praktyki
- możliwość stosowania trudnych i silnych konstrukcji vs. kod łatwy z rozumieniu i utrzymaniu
- błędy wykonania vs. błędy kompilacji
- łatwość prototypowania
- konieczność obchodzenia ograniczeń języka
- możliwość stosowania "dziwnych" i niebezpiecznych rozwiązań ad hoc