Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

XII. Java po C++.
Pakiety, interfejsy i obsługa wyjątków w Javie

1. Pakiety

1.1. Pakiet to swoisty pojemnik na klasy, używany do odpowiedniego podziału przestrzeni nazw oraz sterowania dostępem do tych nazw.

1.2. Tworzenie pakietu odbywa się poprzez umieszczenie instrukcji `package` jako pierwszej instrukcji w pliku źródłowym Javy:

`package` *nazwa_pakietu*

Wszystkie klasy zdefiniowane w pliku zostaną przypisane do pakietu wymienionego w instrukcji. Pominięcie instrukcji `package` jest równoznaczne z przypisaniem klasy do domyślnej przestrzeni nazw.

1.3. Do przechowywania pakietów służy system plików. Każdy plik o rozszerzeniach `.java` i `.class`, który został utworzony dla pakietu o nazwie *nazwa_pakietu*, będzie umieszczony w katalogu o nazwie *nazwa_pakietu*.

1.4. Pakiety można składać w wielopoziomowe hierarchie; w nazwie pakietu poszczególne poziomy oddziela się znakiem kropki, np.

```
package java.awt.image; /* Abstract Window Toolkit */
```

Wyszukiwanie żądanych pakietów odbywa się poprzez przeszukiwanie katalogu bieżącego oraz katalogów zadanych przez zmienną środowiskową `CLASSPATH`.

1.5. Oto przykład prostego pakietu:

```
package MyPack;
class Balance {
  String name;
  double bal;
  Balance(String n, double b) {
    name = n;
    bal = b;
  }
  void show() {
    if(bal<0)
      System.out.print(”-->> ”);
      System.out.println
                 (name + ”: ” + bal + ”zl”);
  }
}
class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];
    current[0] =
          new Balance(”Romeo i Julia”,151.30);
    current[1] =
          new Balance(”Grona gniewu”, 1142.05);
    current[2] =
          new Balance(”Mercedes-benz”,-28.50);
    for(int i=0;i<3;i++) current[i].show();
  }
}
```

Plik źródłowy, o nazwie `AccountBalance.java` zostanie umieszczony w katalogu `MyPack`. Uruchomienie programu nastąpi na podstawie polecenia:

```
java MyPack.AccountBalance
```

w którym nazwa klasy musi być koniecznie poprzedzona nazwą pakietu.

1.6. Pakiety, współdziałając z klasami, tworzą nowe kategorie widoczności składowych klas. Możliwe poziomy dostępu podsumowuje następująca tabela:

	`private`	`brak`	`protected`	`public`
ta sama klasa	**tak**	**tak**	**tak**	**tak**
ten sam pakiet, podklasa	**nie**	**tak**	**tak**	**tak**
ten sam pakiet, inna klasa	**nie**	**tak**	**tak**	**tak**
inny pakiet, podklasa	**nie**	**nie**	**tak**	**tak**
inny pakiet, inna klasa	**nie**	**nie**	**nie**	**tak**

1.7. Za pomocą instrukcji `import` można uwidocznić w programie konkretne klasy lub całe pakiety. Używanie tej instrukcji jest konieczne także w stosunku do wszystkich wbudowanych klas Javy, z wyjątkiem tych umieszczonych w pakiecie `java.lang`. Ogólna postać instrukcji `import` to:

`import` *pakiet1*[`.`*pakiet2*... `.`*pakietn*]`.`(*nazwa_klasy* | *)`;`

Należy ją umieszczać bezpośrednio po instrukcji `package`, przed definicjami klas.

1.8. Gdyby zmienić w zdefiniowanym wyżej pakiecie `MyPack` wybrane modyfikatory dostępu na publiczne:

```
package MyPack;
public class Balance {
  String name;
  double bal;
  public Balance(String n, double b) {
    name = n;
    bal = b;
  }
  public void show() {
    if(bal<0)
      System.out.print(”-->> ”);
      System.out.println
                 (name + ”: ” + bal + ”zl”);
  }
}
```

to będzie można użyć metod `Balance` i `show` w kontekście jak poniżej:

```
import MyPack.*;
class TestBalance {
  public static void main(String args[]) {
    Balance test =
       new Balance(”Program. w C++”, 2451.00);
    test.show();
  }
}
```

Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

2. Interfejsy

2.1. Za pomocą konstrukcji `interface` można oddzielić specyfikację interfejsu klasy od pełnej implementacji tej klasy. Konstrukcja `interface` przypomina pod względem składniowym definicję samej klasy: obejmuje zestaw sygnatur metod (w miejsce samych definicji metod) oraz zestaw zmiennych, które mają charakter finalny (`final`). Jej ogólna postać jest następująca:

modyfikator_dostępu `interface` *nazwa* `{`
    sygnatura1 `;`
    sygnatura2 `;`
    ...
    sygnaturan `;`
    zmienna_finalna1 `;`
    zmienna_finalna2 `;`
    ...
    zmienna_finalnam `;`
`}`

2.2. Po zdefiniowaniu, interfejs może być implementowany przez wiele klas. Poprawna implementacja interfejsu wymaga dodania do klasy klauzuli `implements` oraz zdefiniowania w tej klasie wszystkich metod zadeklarowanych w interfejsie. Metody implementujące interfejs muszą być metodami publicznymi.

2.3. Oto przykład definicji interfejsu i jego implementacji przez pewną klasę:

```
interface Callback {
  void callback(int param);
}
class Client implements Callback {
  public void callback(int p) {
    System.out.println
     (”wywołanie callback() z wartością ” + p);
  }
}
```

2.4. W klasie implementującej interfejs, oprócz definicji metod zadeklarowanych w interfejsie, można umieścić też definicje metod dodatkowych, np.

```
class Client implements Callback {
  public void callback(int p) {
    System.out.println
     (”Wywołanie callback() z wartością ” + p);
  }
  void additionalmethod() {
    System.out.println
     (”To jest własna metoda.”);
  }
}
```

2.5. W programie możliwe jest zadeklarowanie i posługiwanie się zmiennymi referencyjnymi o typie interfejsu. Do takiej zmiennej można przypisać obiekt dowolnej klasy implementującej dany interfejs. Wywołanie metody dla takiej referencji pociąga za sobą konieczność dynamicznego wyszukania jej wersji w trakcie działania programu (uwaga na narzuty czasowe związane z takim wyszukiwaniem!). Należy pamiętać, że zmienna referencyjna „wie” tylko o metodach zadeklarowanych w jej interfejsie.

```
class Testify {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

2.6. Choć zasadniczym przeznaczeniem interfejsu jest definiowanie „szkieletów” metod, które zostaną w odpowiednim miejscu zaimplementowane, to można go użyć także do specyfikacji zmiennych o charakterze stałych, które będą używane przez różne klasy implementujące.

2.7. Interfejsy, podobnie jak same klasy, podlegają dziedziczeniu (rozszerzaniu). Interfejs może rozszerzać jedynie inny interfejs (nie klasę!). O ile jednak klasa może rozszerzać tylko jedną klasę bazową, to interfejs może rozszerzać wiele interfejsów ogólnych. Klasa implementująca interfejs dziedziczący po innych interfejsach musi zapewnić definicje wszystkich metod zadeklarowanych w całym łańcuchu dziedziczenia, np.

```
interface A {
  void method1();
  void method2();
}
interface B extends A {
  void method3();
}
class MyClass implements B {
  public void method1() {
   System.out.println(”Implementacja method1()”);
  }
  public void method2() {
   System.out.println(”Implementacja method2()”);
  }
  public void method3() {
   System.out.println(”Implementacja method3()”);
  }
}
```

2.8. Podsumowując, interfejs przypomina do pewnego stopnia abstrakcyjną klasę polimorficzną. Różni się od niej jednak zasadniczo pod względem składniowym, a także, w innych aspektach (m.in. zmienne finalne) – pod względem semantycznym.

3. Obsługa wyjątków w Javie

3.1. Wszystkie wyjątki muszą być podklasami wbudowanej klasy `Throwable`. Pochodnymi tej klasy, znajdującej się na szczycie hierarchii klas odnoszących się do wyjątków, są dwie podklasy, dzielące wyjątki na dwie kategorie. `Exception` (z podklasą `RuntimeException`) to podklasa opisująca sytuacje wyjątkowe, które aplikacja powinna wychwycić i obsłużyć w trakcie swego działania. `Error` określa wyjątki, które – choć mogą wystąpić – nie powinny być wychwytywane i obsługiwane.

3.2. Wyjątki które nie zostaną obsłużone w sposób programowy wychwyci domyślna procedura obsługi. Niniejsza procedura wyświetla tekst opisujący wyjątek, pokazuje stos wywołań prowadzących do metody zgłaszającej wyjątek i powoduje zakończenie programu, np.

```
class Exc1 {
  static void subroutine() {
    int d=0;
    int a=10/d; System.out.println(a);
  }
  public static void main(String args[]) {
    Exc1.subroutine();
  }
}
```

Stos wywołań generowany przez domyślną procedurę obsługi ma w tym wypadku postać:

```
java.lang.ArithmeticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

3.3. Programowe wychwytanie wyjątków odbywa się przy użyciu znanych z C++, tworzących jedną jednostkę, konstrukcji `try` i klauzuli `catch`. W Javie wszystkie wyjątki muszą być obiektami-egzemplarzami klasy dziedziczącej po

`Throwable`. W związku z tym, argumentem `catch` jest zawsze wyjątek-obiekt, np.

```
import java.util.Random;
class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();
    for(int i=0; i<32999; i++) {
      try {
        b=r.nextInt();
        c=r.nextInt();
        a=12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println
           (”Dzielenie przez zero.”);
        // co nalezy wstawic w tym miejscu?
      }
      System.out.println(”a: ” + a);
    }
  }
}
```

3.4. Wyświetlenie standardowego opisu wyjątku następuje poprzez przekazanie obiektu-wyjątku do metody `println()`. Powyższa klauzula `catch`, zredefiniowana do postaci:

```
catch (ArithmeticException e) {
  System.out.println(”Wyjątek:” + e);
  a=0;   //wartosc domyslna
}
```

da odpowiedź w postaci:

```
Wyjątek: java.lang.ArithmeticException: / by zero
```

3.5. Z jedną konstrukcją `try` można związać wiele klauzul `catch`, pamiętając, by na początku umieszczać te z najbardziej szczegółowymi podklasami, dopiero później – ogólne (kod nieosiągalny jest w Javie kodem błędnym!).

3.6. Konstrukcje `try` można zagnieżdżać; jeśli wewnętrzna konstrukcja `try` nie zawiera stosownej klauzuli `catch`, to Java – przy wykorzystaniu stosu – przekazuje sterowanie do klauzuli `catch` zewnętrznej konstrukcji `try`. Niejawne zagnieżdżanie `try` ma miejsce przy wywoływaniu metod posiadających własny blok `try`.
Jeśli nie zostanie odnaleziona żadna właściwa procedura obsługi wyjątku, Java wykona domyślną procedurę obsługi.

3.7. Java obsługuje nie tylko wyjątki zgłaszane przez jej system wykonawczy. Można w programie zgłosić wyjątek własny, zdefiniowany programowo. Niezależnie od charakteru wyjątku, jego zgłoszenie odbywa się przy użyciu instrukcji w postaci:

`throw` *obiekt-typu-Throwable-lub-pochodnego*`;`

gdzie *obiekt-typu-Throwable-lub-pochodnego* jest obiektem typu `Throwable` lub jedną z jego podklas (inne typy nie wchodzą w rachubę!), np.

```
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException(”demo”);
    } catch(NullPointerException e) {
      System.out.println
          (”Złapane wewn. metody demoproc().”);
      throw e;
    }
  }
```

```
  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println
         (”Ponowne złapanie: ” + e);
    }
  }
}
```

3.8. Jeżeli metoda posiada wyjątek, a nie potrafi go sama obsłużyć, to w definicji tej metody trzeba umieścić specjalną klauzulę `throws` z informacją o tym wyjątku. Uwaga ta nie dotyczy ani wyjątków typu `Error`, ani licznych wyjątków `RuntimeException` (i jej podklas), np.

```
class ThrowsDemo {
   static void throwOne()
                          throws IllegalAccessException {
      System.out.println(”Wewn. throwOne.”);
      throw new IllegalAccessException(”demo”);
   }
   public static void main(String args[]) {
      try {
        throwOne();
      } catch (IllegalAccessException e) {
        System.out.println(”Złapano ” ???);
      }
   }
}
```

Po uruchomieniu tego programu uzyskujemy komunikaty:

???

```
Wewn. throwOne.
Złapano java.lang.IllegalAccesssException: demo
```

3.9. Słowo kluczowe `finally` pozwala zdefiniować blok kodu, który zostanie wykonany po zakończeniu bloku `try-catch`, lecz przed kodem znajdującym się bezpośrednio za blokiem `try`. Blok wykona się zarówno wtedy, gdy z obszaru `try` nie zostanie zgłoszony żaden wyjątek, jak i wtedy, gdy wyjątek ten zostanie zgłoszony (niezależnie od tego, czy za obszarem `try` znajduje się sekcja `catch` pasująca do typu wyjątku, czy nie). Blok `finally` wykona się zawsze, niezależnie od przyczyny opuszczenia obszaru `try`. Jeśli opuszczenie obszaru `try` lub sekcji `catch` następuje na skutek wykonania instrukcji `return`, to blok `finally` wykona się przed zakończeniem programu. Konstrukcja `try` wymaga zdefiniowania przynajmniej jednej klauzuli `catch` lub bloku `finally`.

3.10. Standardowy pakiet `java.lang` definiuje kilka klas wyjątków. Najbardziej ogólne są podklasami klasy `RuntimeException`. Ze względu na to, że pakiet `java.lang` jest niejawnie importowany do wszystkich programów Javy, większość wyjątków wywodzących się z tej klasy jest dostępna automatycznie. Wyjątków tych, zwanych nieweryfikowanymi, nie trzeba też umieszczać na liście klauzuli `throws` (np. `NullPointerException()`). W pakiecie `java.lang` są jednak także wyjątki weryfikowane, które – przy braku ich obsługi w metodzie – wymagają umieszczania na liście `throws` (np. `IllegalAccessException()`).

3.11. W Javie można też tworzyć własne podklasy wyjątków. Wystarczy w tym celu rozszerzyć klasę `Exception` (podklasę `Throwable`). Ta nowa podklasa nie musi niczego implementować: wystarczy ją umieścić na właściwym miejscu w hierarchii typów. Sama klasa `Exception` dziedziczy wszystkie metody po klasie `Throwable` (nie posiada własnych). Oto przykład:

```
class MyException extends Exception {
  private int detail;
  MyException(int a) {
    detail = a;
  }
  public String toString() {
    return ”MyException[” + detail + ”]”;
  }
}
class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println(”Wywolanie compute(” + a + ”)”);
    if(a > 10)
      throw new MyException(a);
    System.out.println(”Normalne wyjście”);
  }
  public static void main(String args[]) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println(”Złapano ” + e);
    }
  }
}
```

Wynikiem działania programu będzie:

```
Wywołanie compute(1)
Normalne wyjście
Wywołanie compute(20)
Złapano MyException[20]
```

3.12. W wersji J2SE języka wprowadzono konstrukcję programową dotyczącą obsługi wyjątków. Są to tzw. łańcuchy wyjątków. Aby umożliwić ich tworzenie, do klasy szczytowej `Throwable` dodano dwa konstruktory (przeciążenie) oraz dwie nowe metody: `getCause()` i `initCause(Throwable` *wyjwcześn*`)`. Wynikiem

pierwszej z nich jest wyjątek położony w łańcuchu o pozycję wyżej, zaś drugiej – referencja do powiązanych ze sobą: wyjątku aktualnego i wyjątku wcześniejszego.