Programowanie komputerów

Wykład 8: „Java - szczegóły technologiczne”

dr inż. Walery Susłow
swalover@ie.tu.koszalin.pl

Wykorzystanie plików JAR

Do czego stosuje się pliki JAR?

- Archiwum JavaTM (JAR) jest to format, który umożliwia nam łączyć wieloplikowe projekty do pojedynczego pliku archiwum.

- Typowo archiwum JAR zawiera pliki *.class i wspomagające pliki połączone z appletami i aplikacjami.

Zalety JAR

- Bezpieczeństwo: możemy podpisać cyfrowo zawartość pliku JAR. Użytkownicy, którzy rozpoznają podpis mogą potem opcjonalnie dowolnie przyznawać przywileje bezpieczeństwa oprogramowania.

- Zmniejszanie czasu załadowania: jeżeli aplet połączony jest w JAR pliki, to klasy i powiązane zasoby mogą być załadowane do przeglądarki w pojedynczej transakcji HTTP bez konieczności otwarcia nowego połączenia dla każdego pliku.

Zalety JAR, cd.

- Kompresja: format JAR pozwala zmniejszać objętość plików dla bardziej efektywnego przechowywania.

- Pakowanie dla rozszerzeń (od wersji 1.2): *struktura rozszerzeń* dostarcza mechanizm przez który możemy dodawać funkcjonalność do rdzenia platformy Java, a format plików JAR określa pakowanie dla rozszerzeń.
  - Java 3D TM i JavaMail są przykładami rozszerzeń rozwijanych przez Sun.
  - Poprzez wykorzystanie formatu plików JAR można udostępniać własne oprogramowanie jako rozszerzenie.

Zalety JAR, cd.

- Opieczętowane pakiety (wer. 1.2): pakiety umieszczone w plikach JAR mogą być opcjonalnie opieczętowane, tak że pakunek może wymusić trwałość wersji.
  - Opieczętowanie pakietu w pliku JAR oznacza, że wszystkie klasy zdefiniowane w danym pakiecie mają znajdować się w tym samym pliku JAR.
- Wersjonowanie pakietu (wer. 1.2): plik JAR może trzymać dane dotyczące zawartych w nim plików, takie jak nazwę sprzedawcy i numer wersji.
- Przenośność: mechanizm pracy z plikami JAR jest standardową częścią rdzenia API platformy Java.

Tworzenie pliku JAR

- Pliki są pakowane do JAR w standardzie ZIP, tak że na nich można wykonywać typowe ZIP-zadania: kompresję, archiwizację, dekompresję, rozpakowanie archiwum.

- Do wykonania podstawowych zadań na plikach JAR wykorzystujemy „Java Archive Tool”, prowadzone jako część JDK. Narzędzie to jest przywoływane przez używanie rozkazu jar.

Operacja	Polecenie
• Utwórz plik JAR	jar cf jar-file <input-file(s)>
• Przeglądaj zawartość pliku JAR	jar tf <jar-file>
• Wyodrębnij zawartość pliku JAR	jar xf <jar-file>
• Wyodrębnij wskazane pliki z archiwum JAR	jar xf jar-file <archived-file(s)>
• Uruchom aplikację spakowaną do JAR (1.1)	jre -cp app.jar <MainClass>
• Uruchom aplikację spakowaną do JAR (1.2, wymagany jest „Main-Class manifest header”)	java -jar <app.jar>
• Przygotuj aplet w postaci JAR do uruchomienia na stronie www	<applet code = AppletClassName.class archive="JarFileName.jar" width=width height=height> </applet>

Interfejsy

Interfejs

- Słowo kluczowe interface produkuje klasę całkowicie abstrakcyjną, nie zawierającą żadnej implementacji
- Interfejs to jednak coś więcej niż klasa abstrakcyjna, umożliwia ona bowiem „wielokrotne dziedziczenie” – można stworzyć klasę, którą będzie można rzutować na więcej niż jeden typ bazowy

Interfejsy – przykład

```
interface DajacyGlos{
        void dajGlos(); //zawsze publiczna
}
class Wrobel implements DajacyGlos{
        public void dajGlos(){
                System.out.println("Ćwir Ćwir");
        }
}
class Pianino implements DajacyGlos{
        public void dajGlos(){
                System.out.println("Do re mi");
        }
}
public class TestGlosu{
        public static void main(String[] args){
                DajacyGlos spiewak[] = {
                        new Wrobel(), new Wrona(), new Pianino()};
                for (int i=0;i<spiewak.length;i++)
                        spiewak[i].dajGlos();
        }
}
```

„Wielokrotne dziedziczenie”

- Każda klasa w Java może dziedziczyć z najwyżej jednej klasy, ale może implementować wiele interfejsów
- Obiekty takiej klasy można rzutować zarówno na klasę bazową, jak i na każdy z zaimplementowanych interfejsów

Rozszerzanie interfejsu

```
interface Potwor{
  void postrasz();
}
interface GroznyPotwor extends Potwor{
  void zniszcz();
}
interface Killer{
  void zabij();
}
interface Wampir extends GroznyPotwor, Killer{
  void wypijKrew();
}
```

Klasy wewnętrzne

Klasy wewnętrzne

- Można umieścić definicję klasy wewnątrz innej klasy (mogą być nawet tworzone wewnątrz metod) – taka klasa nazywa się klasą wewnętrzną
- Klasy wewnętrzne umożliwiają grupowanie logicznie powiązanych ze sobą klas i kontrolowanie widoczności jednych w drugich

Klasy wewnętrzne – przykład

```
class Zewn{
  class Wewn1{
  private int wart = 11;
  public int getWart(){return wart;}
  }
  class Wewn2{
  private String napis;
  Wewn2(String s){napis=s;}
  String getNapis(){return napis;}
  }
  public void test(String s){
  Wewn1 w1 = new Wewn1();
  Wewn2 w2 = new Wewn2(s);
  System.out.println(w2.getNapis());
  }
}
```

Tworzenie obiektów klas wewnętrznych

- Stworzenie obiektu klasy wewnętrznej przebiega tak samo jak dowolnej innej klasy:

```
Wewn1 w1 = new Wewn1();
```

- Jeśli chcemy stworzyć obiekt klasy wewnętrznej, a jesteśmy poza klasą zewnętrzną to do nazwy klasy wewnętrznej odwołujemy się przez nazwę klasy zewnętrznej:

```
Zewn.Wewn1 w1 = new Zewn.Wewn1();
```

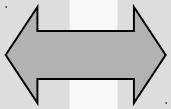
Zasięg zmiennych

```
//Klasa wewnętrzna ma dostęp do pól i metod
//klasy zewnętrznej, w której jest umieszczona
class Zewn{
  private int zmienna=4;
  class Wewn1{
  private int wart = 11+zmienna;
  public int getWart(){return wart;}   // 15
  }
}
```

Klasy anonimowe (1)

Istnieje możliwość tworzenia klas bez nadawania im nazwy. Taka klasa musi dziedziczyć z innej (lub implementować jakiś interfejs) i może jedynie przesłaniać metody klasy bazowej (implementować metody interfejsu).

```
class Anonimowa
   extends Bazowa{
   int metoda(int arg){
       //ciało metody
   }
}
Bazowa obiekt = new
   Anonimowa();
```



```
Bazowa obiekt = new
   Bazowa(){
   int metoda(int arg){
       //ciało metody
   }
};
```

Klasy anonimowe (2)

```
//Zamyka okno dialogowe, gdy użytkownik kliknie przycisk „Odrzuć”
dismissButton.addActionListener ( new ActionListener())
    {
     public void actionPerformed ( ActionEvent event )
       {
       Object object = event.getSource();
       if ( object == dismissButton )
          {
          dismiss();
          } // end if
       } // end actionPerformed
    } // end anonymous class
);   // end addActionListener line
```

Klasy anonimowe (3)

```
// To samo z klasą nazwaną
dismissButton.addActionListener( new CustomListener() );
...
private static class CustomListener implements ActionListener
  {
    public void actionPerformed ( ActionEvent event )
     {
     Object object = event.getSource();
     if ( object == dismissButton )
       {
       dismiss();
       }
     }
  }
```

Lokalizacja programu Java

Lokalizacja (1)

- Lokalizacja to specyficzne dla danego języka, kraju i regionu reguły dotyczące prezentacji różnych informacji (formatowania liczb, formatowania dat, pisowni tekstów, porządku alfabetycznego, itp).
- Programy powinny być dostosowane do różnych lokalizacji bez ponownego kompilowania kodu.

Lokalizacja (2)

- W pakiecie **java.text** mamy cały zestaw klas rozwiązujący zagadnienia lokalizacyjne.
- Lokalizacja w Javie jest reprezentowana przez klasę **Locale** z pakietu **java.util**.
- Lokalizacja jest określana przez kombinację kodu języka, kraju i wariantu (regionu):
  - Kod języka to dwuliterowy skrót (małe litery) wg standardu ISO-639: http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt
  - Kod kraju to dwuliterowy skrót (duże litery) wg standardu ISO-3166: http://ftp.ics.uci.edu/pub/ietf/http/related/iso3166.txt
  - Kod wariantu (regionu) to dodatkowa informacja, która nie spełnia określonych standardów.

Lokalizacja (3)

- Lokalizacja w Javie jest określana przez kombinację kodu języka, kraju i wariantu (regionu):
  - Locale (String lang)
  - Locale (String lang, String coun)
  - Locale (String lang, String coun, String var)
- Przykłady obiektów lokalizacyjnych:
  - Locale a = new Locale("pl","PL");
  - Locale b = new Locale("en","GB");
  - Locale c = new Locale("en","US");
  - Locale d = new Locale("en");
- Każdy program w Javie uzyskuje domyślną lokalizację na podstawie właściwości ustawionych dla platformy systemowej.
- Bieżącą lokalizację możemy odczytać za pomocą:
  Locale.getDefault();
- Domyślną lokalizację można zmienić za pomocą:
  Locale.setDefault(newLocale);

Lokalizacja (4)

- Klasa Locale posiada kilka użytecznych getterów:
  - String getLanguage()
  - String getCountry()
  - String getVariant()
- Klasa Locale posiada kilka metod prezentacyjnych:
  - String getDisplayLanguage();
  - String getDisplayCountry();
  - String getDisplayVariant();
  - String getDisplayName();
- Klasa Locale posiada kilka informacyjnych metod statycznych:
  - String[] getISOLanguages()
  - String[] getISOCountries()
  - Locale[] getAvailableLocales()

Lokalizacja (5)

- Klasy lokalizacyjnie czułe biorą pod uwagę lokalizację – zaliczamy do nich:
  - NumberFormat (formatowanie liczb)
  - DateFormat (formatowanie dat)
  - Calendar (data i czas)
  - Collator (porządek alfabetyczny)
  - BreakIterator (rozbiór tekstu)

Formatowanie

- Do formatowania liczb (całkowitych i zmiennopozycyjnych) służą klasy **NumberFormat** i **DecimalFormat**.
- Klasa **Currency** opisuje waluty.
- Strefy czasowe opisuje klasa **TimeZone**.
- Informację o dacie i godzinie przechowuje się w obiekcie **Calendar** lub **Date**, a do formatowania czasu używa się klasy **DateFormat**.

Waluta

java.util

Class Currency

java.lang.Object

java.util.Currency

public final class Currency

extends Object

implements Serializable

getAvailableCurrencies()

GetCurrencyCode()

getDefaultFractionDigits()

getDisplayName()

getDisplayName(Locale locale)

getInstance(Locale locale)

getInstance(String currencyCode)

getNumericCode()

getSymbol()

getSymbol(Locale locale)

toString()