Higher Order Programing through Java Reflection

Autor: Konrad Witkowski

Wprowadzenie

- programowanie wyższego rzędu jest cechą charakterystyczną języków funkcyjnych
- funkcje są „obywatelami pierwszej kategorii” czyli mogą być parametrami funkcji, zwracane jako wartości wyliczone, mogą być wartościami w strukturach danych, etc.

Plusy i minusy

Plusy

- siła wyrazu
- zwięzłość
- „czystość” i dobra struktura
- łatwość ponownego wykorzystania kodu
- mniejsza liczba błędów

Minusy

- niska wydajność implementacji

Kompromis

Pewne elementy programowania wyższego rzędu zostały dodane do języków imperatywnych (C,Pascal,C++)

Programowanie wyższego rzędu w językach obiektowych

Kolekcje figur geometrycznych wraz z metodami do obliczania pól i obwodów :

```
public class FList {
   private Object elem;
   private FList next;
   private int sz;
   public FList() { sz = 0; }
   public void insert(Object x){
        if (sz==0){
                elem = x;
                sz = 1;
                next = new FList();
        } else {
                sz++;
                next.insert(x);
        }
        }
        public Object val() { return elem; }
        public FList tail() { return next; }
        public int size() { return sz; }
}
```

```
/* klasa reprezentująca obiekty geometryczne */
public abstract class Shape {
      public abstract double area();
      public abstract double perimeter();
}
/* klasa reprezentująca prostokąty */
public class Rectangle extends Shape {
      private double base;
      private double height;
      public Rectangle(double b, double h)     {
            base = b;
            height = h;
      }
      public double area() { return base*height; }
      public double perimeter() {
            return 2*base + 2*height;
      }
}
```

```
public class ShapeList extends FList {
   public FList areaList(){
     FList L = new FList();
     if (size()!=0){
       L = ((ShapeList)tail()).areaList();
       L.insert(new Double(((Shape)val()).area()));
     }
     return L;
   }
   public FList perimeterList(){
     FList L = new FList();
     if (size()!=0){
       L = ((ShapeList)tail()).perimeterList();
       L.insert(new
            Double(((Shape)val()).perimeter()));
     }
     return L;
   }
}
```

Rozwiązanie nr 1

```
public class FList{
/* zmienne i metody jak wyżej */
   public FList map(Method F){
       FList L = new FList();
       if (size()!=0){
             L.insert(val().F());
       }
       return L;
   }
   public FList areaList() {
       return map(area);
   }
   public FList perimeterList() {
       return map(perimeter);
   }
}
```

Reflection

Class – klasa wszystkich klas (definicja w java.lang)

Method – klasa wszystkich metod statycznych i instancyjnych (java.lang.reflect)

Class getClass() – metoda obiektu, która zwraca jego klasę

Method getMethod(String *name,****Class*** **[]** *paramType)* – metoda obiektów klasy ***Class***, która zwraca metodę o nazwie *name* i parametrach odpowiadających *paramType*.

Object invoke(Object *obj,****Object*** **[]** ***args)*** – metoda obiektów klasy ***Method***, wykonuje metodę na obiekcie *obj* z parametrami *args*.

Rozwiązanie 2

```
public class FList{
/* zmienne i metody jak wyżej */
  public FList map(String F)    {
      Object [] Arg = {};
      FList L = new FList();
      try{
                  if (size()!=0){
                  L = tail().map(F);
                  Method M = val().getClass().getMethod(F,Arg);
                  L.insert(M.invoke(val(),Arg));
            }
      }
      catch (Exception e) {}
      return L;
  }
}
```

Rozwiązanie 3 – metody statyczne

```
public class FList {
/* zmienne i metody jak wyżej */
public FList map(Method f){
   FList L = new FList();
   try{
        if (size()!=0){
               L = tail().map(f);
               Object [] Arg = {this.val()};

        L.insert(f.invoke(null,Arg));
        }
   }
   catch (Exception e) {}
   return L;
   }
}
```

```
public class ShapeList extends FList {

        public static double myArea(Shape s)
        {
                return s.area();
        }

        public FList areaList() throws
                NoSuchMethodException,ClassNotFoundException
        {
                Class [] Arg = {Class.forName("Shape")};
                return map(getClass().getMethod("myArea",Arg));
        }

}
```

Inne rozwiązania

Anonimowe klasy wewnętrzne

- Idea : emulowanie funkcji przez obiekty pewnych klas
- każda funkcja funkcja jest reprezentowana przez jedną klasę
- każda funkcja ma unikalną metodę *apply*, której wywołanie powoduje wykonanie funkcji

Rozwiązanie 4

```
public interface Compute{
   Object app(Object x);
}
public class FList{
 public FList map(Compute m) {
   FList L = new FList();
   try{
       if (size()!=0){
          L = tail().map(m);
          L.insert(m.app(this.val()));
       }
   }
   catch (Exception e) {
   e.printStackTrace(); }
   return L;
  }
}
```

```
public class Shape{
    public static Compute computeArea()
    {
        return new Compute(){
            public Object app(Object x) { return null; }
        };
    }
}

public class Rectangle extends Shape{
    private double base;
    private double height;
        public Rectangle(double b,double h)
                { base = b; height = h; }
        public static Compute computeArea(){
        return new Compute(){
            public Object app(Object x){
                Rectangle r = (Rectangle) x;
                return new Double(r.base*r.height);
            }
        };
    }
}
```

Wywołanie : `LL = L.map(Shape.computeArea());`

Wady rozwiązania

- metoda jest wyznaczana statycznie
- argumentami tak naprawdę nie są metody tylko obiekty je reprezentujące. Tak więc tylko metody, które mają zadeklarowane odpowiednie klasy mogą być przekazywane jako parametry do metod wyższego rzędu

Delegaty

Delegaty to obiekty reprezentujące metody (środowisko Visual J++).

Możliwe są dwa rodzaje wiązań – wczesne (early-bound) w momencie stworzenia delagata, oraz późne (late-bound) w czasie wykonania.

Rozwiązanie 5a (early-bound)

```
delegates Object Fun(Object o);
public FList map(Fun m)
{
   FList L = new FList();
   try{
        if (size()!=0) {
               L = tail().map(m);
               L.insert(m.invoke(this.val()));
        }
   }
   catch (Exception e) {}
   return L;
}
public class Shape{
   public static Object myArea(Shape s) {
        return null;
   };
}
```

```
public class Rectangle extends Shape {
        private double base;
        private double height;
        public static Object myArea(Rectangle r) {
                return new Double(r.base*r.height);
        }
}

...

LL = L.map(new Fun(Shape.myArea));

...
```

Rozwiązanie 5b (late-bound)

```
delegates Object Fun();
public FList map(String s) {
   FList L = new FList();
   try {
       if (size()!=0) {
              L = tail().map(s);
              Fun m = new Fun(this.val(),s);
              L.insert(m.invoke());
       }
   }
   catch (Exception e) {}
   return L;
}
```

Pizza

Pizza jest to rozszerzenie języka Java, które dostarcza takich mechanizmów jak typy parametryzowane, abstrakcja funkcji, *pattern matching*.

Typy parametryzowane

```
public class Pair<A,B>
{
    public A fst;
    public B snd;
    public Pair(A fst,B snd) {
      this.fst = fst;
      this.snd = snd;
    }
    public A getFirst() { return fst; }
    public B getSecond() { return snd; }
}
```

Abstrakcja funkcji

Składnia typu funkcyjnego:

(*argtype*, ..., *argtype*) -> *resulttype*

lub, jeżeli motoda może „rzucać” wyjątki :

(*argtype*, ..., *argtype*) throws *exception*, ...,
   -> *resulttype*

Przykład :

```
class PrintSortedStrings
{
   static void print(String [] args, (String,String) -> int compare)
     ...
           if (compare(args[i],args[j])>0)
     ...
}
```

Funkcje anonimowe

Pizza umożliwia także tworzenie funkcji anonimowych.

Składnia :

```
fun(arguments) -> resulttype { statements }
```

Ciekawy przykład

```
()->int makeIncr() {
  int count = 0;
  return fun()->int { return count++; };
    }

()->int makeIncr(int start) {
  int count = start;
  return fun()->int { return count++;};
    }

()->int incr = makeIncr();
```

Wywołanie : incr();

Większy przykład

```
class PrintSortedStrings
{
    static void print(String [] args,(String,String) -> int
        compare)
    {
   String s;
   for (int i=0;i<args.length-1;i++)
        for (int j=i+1;j<args.length;j++)
              if (compare(args[i],args[j])>0){
                    s = args[i];
                    args[i] = args[j];
                    args[j] = s;
        }
   for (int i=0;i<args.length;i++)
        System.out.println(args[i]);
  }
}
```

```
public class Test
{
    static int myCompare(String s1,String s2)
    {
        return s1.compareTo(s2);
    }

    static int myCompareNoCase(String s1, String s2)
    {
        return s1.toLowerCase().compareTo(s2.toLowerCase());
    }

    public static void main(String [] args)
    {
        String myStrings [] = { "This", "Strings", "will",
                        "be", "sorted", "!" };

        PrintSortedStrings.print(myStrings,myCompare);
        PrintSortedStrings.print(myStrings,myCompareNoCase);
        PrintSortedStrings.print(myStrings,
                fun(String x, String y)->int {return 1;});

    }
}
```

Koniec.