Interfejsy

Aplikacje implementowane w obiektowych językach programowania konstruowane są w postaci zbioru komunikujących się ze sobą obiektów. Współdziałanie pomiędzy obiektami polega na wzajemnym wywołaniu metod lub korzystaniu z wartości atrybutów.

Wymiana informacji pomiędzy elementami aplikacji ma postać *protokołu*, który określa, jakie metody można wywołać lub, do jakich atrybutów można uzyskać dostęp.

- Z punktu widzenia uczestnika protokołu nie jest istotne, w jaki sposób są implementowane metody drugiej strony. Ważnym jest, aby druga strona zapewniała pewien określony przez zasady współdziałania zbiór metod.
- Specyfikacja protokołu najczęściej abstrahuje od konkretnych struktur danych, które są używane przy implementacji atrybutów. Dostęp do atrybutów może zostać opakowany przez metody `set()` oraz `get()`, które ustawiają lub odczytują ich wartości.

Pod pojęciem *interfejs*, rozumie się zazwyczaj zbiór metod, które implementuje dana klasa i które mogą zostać wywołane z zewnątrz.

Projektując i implementując klasę, który dla realizacji stawianych jej zadań musi współpracować z innym obiektem zakłada się, że należy on do pewnej klasy implementującej określony interfejs.

Przykład

```
abstract class UserInterface
{
  public abstract void showProgress(double p);
  public abstract boolean isAbort();
}

class Processing
{
  void run(UserInterface ui){
    for(int i = 0;i<MAX;i++){
      processElement(i);
      ui.showProgress((double)i/(double)MAX);
      if(ui.isAbort())break;
    }
  }
}
```

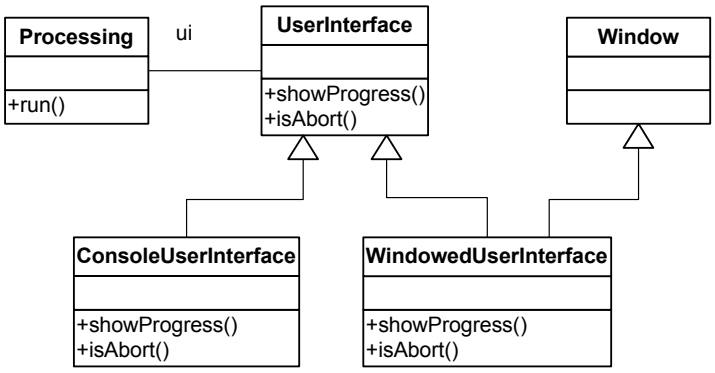
- Klasa `Processing` współpracuje z obiektem klasy `UserInterface`. Ta ostatnia jest klasą abstrakcyjną definiującą interfejs funkcjonalny: `showProgress(double p)` oraz `isAbort()`.
- Wywołując metodę `run()` klasy `Processing` należy przekazać referencję do konkretnej klasy, np.: `ConsoleUserInterface`:

```
Processing p = new Processing ();
p.run(new ConsoleUserInterface());
```

- Klasa `ConsoleUserInterface` jest zdefiniowana jako:

```
class ConsoleUserInterface extends UserInterface
{
  public void showProgress(double p) {
    // wypisz p*100
  }
  public boolean isAbort()  {
    // sprawdz, czy naciśnieto ESC
    return false;
  }
}
```

Dla aplikacji działającej w środowisku okienkowym stworzona zostanie klasa `WindowedUserInterface`. Jej zadaniem jest wyświetlanie okna pokazującego stan zaawansowania przetwarzania oraz umożliwienie zatrzymanie przetwarzania, np.: przez wybranie przycisku *Cancel*.



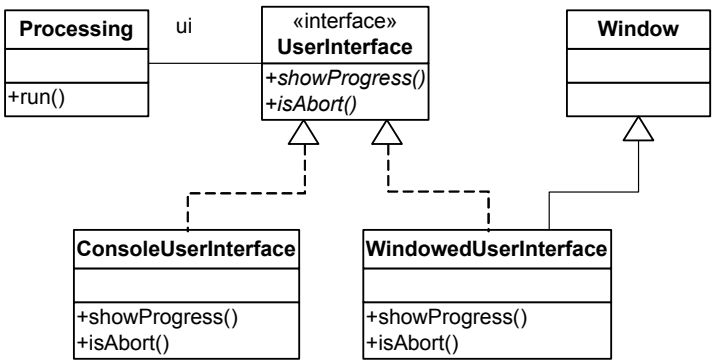
Zastosowanie interfejsu okienkowego wymaga dziedziczenia po klasie należącej do biblioteki obsługującej GUI – np.: `Window`, `Dialog`, `Frame` itd.

Taka konstrukcja jest dopuszczalna w języku zezwalającym na wielokrotne dziedziczenie, np.: C++. Jest jednak niedopuszczalna w języku zakładającym wyłącznie dziedziczenie jednobazowe.

Elementem języka Java umożliwiającym realizację powyższego wzorca jest interfejs (ang. *interface*), który może być traktowany jako rodzaj abstrakcyjnej klasy nie należącej do wbudowanej hierarchii obiektowej.

Klasa należąca do hierarchii obiektowej powiązana jest relacją dziedziczna ze swoją bezpośrednią nadklasą. Równocześnie klasa może implementować (realizować) pewien zbiór interfejsów.

Przykład



- Klasa `ConsoleUserIterface` *implementuje* interfejs `UserInterface` i domyślnie dziedziczy po klasie `Object`.
- Klasa `WindowedUserInterface` *implementuje* interfejs `UserInterface` i dziedziczy po klasie `Window`.
- Klasa `Processing` posługuje się referencją do interfejsu `UserInterface` i za jej pośrednictwem woła metody `showProgress()` i `isAbort()`.

```
interface UserInterface
{
  void showProgress(double p);
  boolean isAbort();
}
class ConsoleUserInterface
implements UserInterface
{
  public void showProgress(double p) {…}
  public boolean isAbort()  {…}
}
class WindowedUserInterface
extends Window
implements UserInterface
{
  public void showProgress(double p) {…}
  public boolean isAbort()  {…}
}
```

Do metody `void run(UserInterface ui)` klasy `Processing` przekazujemy referencję do obiektu klasy `ConsoleUserInterface` lub `WindowedUserInterface` dokonując równocześnie rzutowania *w górę* hierarchii interfejsów/klas.

Wewnątrz metody `run` wywoływane są polimorficzne metody należące do interfejsu `UserInterface: showProgress()` oraz `isAbort()`.

Podstawowe cechy interfejsów

- Interfejsy mogą deklarować wyłącznie metody abstrakcyjne (do Java 1.7 włącznie). Domyślnie metody te są publiczne.
- Interfejsy mogą definiować stałe (atrybuty typu `static final`)
- Deklarując interfejs, wprowadzamy nowy typ danych – referencję do interfejsu. Referencje te mogą być użyte jako atrybuty klas, parametry metod oraz zmienne lokalne.
- Pomiędzy interfejsami może występować relacja dziedziczenia (słowo kluczowe `extends`). Dany interfejs może dziedziczyć po kilku interfejsach. W tym przypadku możliwe jest dziedziczenie wielobazowe.
- W programie można zaimplementować wiele hierarchii interfejsów. Klasy mogą implementować wiele interfejsów należących do różnych hierarchii, ale mogą dziedziczyć po dokładnie jednej nadklasie.

Interfejsy czy klasy abstrakcyjne?

Interfejsy są stosowane z dwóch głównych powodów:

- aby można było zadeklarować klasę należącą do dwóch lub więcej hierarchii typów;
- aby podobnie, jak w przypadku klas abstrakcyjnych, ustalić pewien interfejs programistyczny, który pozwoli na wywołanie polimorficznych metod.

Projektując oprogramowanie w języku Java należy tam, gdzie jest to możliwe, raczej używać interfejsów zamiast abstrakcyjnych klas bazowych.

- Użycie abstrakcyjnej klasy bazowej uniemożliwia późniejsze dowiązanie jej klas potomnych do innej hierarchii dziedziczenia.
- Bazowe klasy abstrakcyjne (lub konkretne) powinny natomiast być stosowane tam, gdzie zachodzi konieczność dzielenia atrybutów oraz dzielenia kodu.

Deklaracja interfejsu

Deklaracja ma postać

InterfaceModifiers$_{opt}$ `interface` *Identifier ExtendsInterfaces*$_{opt}$ *InterfaceBody*

- *InterfaceModifiers*$_{opt}$ – jeden z modyfikatorów: `public`, `protected`, `private`, `static`, `abstract`, `strictfp`. Modyfikatory `protected`, `private` i `static` mogą się pojawić wyłącznie w przypadku zagnieżdżonych interfejsów (zdefiniowanych wewnątrz klasy lub innego interfejsu).
- *Identifier* – nazwa typu (unikalna wewnątrz pakietu)
- *ExtendsInterfaces*$_{opt}$ – deklaracja bezpośrednich nadinterfejsów.
- *InterfaceBody* – ciało interfejsu. Deklaruje składowe.

Składowe interfejsu

Składowymi są:

- deklaracje stałych,
- abstrakcyjnych metod
- zagnieżdżonych klas i interfejsów
- standardowe implementacje metod (od Java 1.8)
- implementacje metod statycznych (od Java 1.8)

Składowe mogą być zadeklarowane w ciele interfejsu lub odziedziczone po nadinterfejsach.

Stałe

Każde pole interfejsu jest zmienną typu `public, static` i `final`, stąd deklaruje stałą. Jawne użycie modyfikatorów nie jest wymagane. Zgodnie z konwencją dla stałych, zaleca się, aby ich identyfikatory były pisane dużymi literami.

Przykład:

```
interface WeekDays {
  int MONDAY =1, TUESDAY=2, WEDNESDAY = 3,   THURSDAY=4,
FRIDAY=5, SATURDAY=6, SUNDAY=7; }
```

Deklaracjom stałych musi towarzyszyć inicjalizacja, ale nie jest wymagane, aby z prawej strony pojawiało się wyrażenie stałe. Możliwe jest odwołanie do wcześniej zadeklarowanych pól.

Przykład:

```
interface WeekDays
{
  int MONDAY =1,
  TUESDAY=MONDAY+1,
  WEDNESDAY = TUESDAY+1,
  THURSDAY=WEDNESDAY+1,
  FRIDAY=THURSDAY+1,
  SATURDAY=FRIDAY+1,
  SUNDAY=SATURDAY+1;
}
```

Przykład:

```
interface NonConstInitialization
{
  java.util.Random rand
    = new java.util.Random();
  double X = rand.nextDouble();
  double Y = rand.nextDouble();
}

public static void main(String[] args) {
System.out.println(NonConstInitialization.X);
System.out.println(NonConstInitialization.Y);
}
```

Dziedziczenie stałych

Interfejs dziedziczy deklaracje stałych po swoim nadinterfejsie:

Przykład

```
interface BaseColors
{
int RED =1, GREEN=2, BLUE = 4;
}
interface PrintColors extends BaseColors
{
  int YELLOW=8, CYAN = 16, MAGENTA =32;
}
```

`PrintColors.RED` jest poprawną stałą interfejsu `PrintColors`. Dla porównania – w języku C++ stałe preprocesora są widoczne od momentu deklaracji; natomiast deklaracje typu `enum` nie są dziedziczone i należy do nich odwoływać się jako `BaseColors::RED`, a nie `PrintColors::RED`.

Przesłanianie. Jeżeli interfejs deklaruje pole (stałą) o takiej samej nazwie, jak pole odziedziczone, wówczas deklaracja *przesłania* deklarację w nadinterfejsie.

Przykład

```
interface A { int A_CONSTANT=1; }
interface B extends A
{ int A_CONSTANT=2, B_CONSTANT=4;}
```

Stała `B.A_CONSTANT` ma wartość 2 i przesłania `A.A_CONSTANT` (wartość 1).

Kolizja nazw. Jeżeli interfejs dziedziczy stałe o tych samych nazwach, wówczas dochodzi do kolizji nazw (niejednoznaczności), co jest raportowane jako błąd kompilacji.

Metody

Interfejs może deklarować metody. Metody te są domyślnie typu `abstract` i `public` (użycie modyfikatorów nie jest wymagane).

Składnia deklaracji metod:

Modifiers$_{opt}$ *ResultType MethodDeclarator Throws*$_{opt}$;

Modifiers$_{opt}$ – `public` lub `abstract`

ResultType – zwracany typ

MethodDeclarator – deklaracja (sygnatura) metody

Throws$_{opt}$ – lista wyrzucanych wyjątków

Do wersji Java 1.7 interfejs mógł deklarować jedynie metody abstrakcyjne, stąd nie występowała ich implementacja. Taka praktyka jest wciąż najczęściej spotykana.

Metody nie mogą mieć modyfikatorów wskazujących na ich implementację, np.: `synchronized`, `native`, `strictfp`. Natomiast modyfikatory te mogą pojawić się w deklaracjach w klasach, która realizuje (implementuje) interfejs.

Zbiór metod interfejsu może być rozszerzany przez dziedziczenie.

Przykład

```
interface IA
{
 void f();
}
interface IB extends IA
{
 int g();
}
```

Interfejs `IB` dziedziczy po `IA` metodę `f()` i dodaje metodę `g()`.

Przedefiniowywanie metod. Jeżeli interfejs deklaruje metodę o takiej samej sygnaturze, jak metoda odziedziczona, wówczas ją *przedefiniowuje*. Wymagana jest wówczas zgodność zwracanych typów oraz wyrzucanych wyjątków.

Kolizja nazw metod. Interfejs może odziedziczyć po dwóch lub więcej nadinterfejsach metody o takiej samej sygnaturze. Jest wówczas również wymagana zgodność zwracanych typów oraz listy wyrzucanych wyjątków.

Przykład niestandardowy

Zazwyczaj klasa realizująca interfejs dostarcza implementacji metod wyspecyfikowanych w interfejsie. Poniższy przykład pokazuje sytuację, w której klasa `B` dziedziczy implementację metody o sygnaturze określonej w interfejsie `I`.

Przykład

```
interface I
{
 void foo();
}
class A
{
  public void foo(){
  System.out.println("foo()");
  }
}
class B extends A implements I { }
class Test{
  public static void main(String[] args) {
    new B().foo();
  }
}
```

Deklaracje typów w interfejsie

Wewnątrz interfejsu można zadeklarować typy (klasy lub interfejsy). Niejawne, typy te są zadeklarowane jako `static` oraz `public`. Jeżeli nazwą interfejsu jest `I`, natomiast nazwą typu `C`, wówczas pełną nazwą typu wewnętrznego jest `I.C`.

Zazwyczaj typy wewnętrzne wprowadza się dla zdefiniowania danych, które będą wyłącznie wykorzystywane jako argumenty metod interfejsu.

Przykład

```
interface IInnerClass
{
 class A  {
 void foo(){System.out.println("A->foo()");}
 }
 void useA(A a);
}
class ITest implements IInnerClass
{
 public void useA(A a)  {
   a.foo();
 }
}
public class Test {
  public static void main(String[] args) {
    new ITest().useA(new IInnerClass.A());
  }
}
```

Wypisze: `A->foo()`.

Standardowe implementacje metod interfejsów (default)

W wersji Java 1.8 wprowadzono możliwość zapewnienia standardowej implementacji metod interfejsów. Zostanie ona automatycznie użyta, w przypadku kiedy klasa nie zapewni własnej implementacji metody.

- Zaletą jest możliwość dzielenia niejednokrotnie złożonego kodu.
- Można w niej odwoływać się do innych metod (ale dalej interfejs nie ma atrybutów).

Dlaczego użycie standardowych, statycznych a nawet prywatnych metod w interfejsie jest metodologicznie poprawne?

- W odróżnieniu od klas, interfejs nie ma stanu – nie może deklarować atrybutów, których wartość możnaby zmienić.
- Zaimplementowane metody interfejsu nigdy nie zmodyfikują jego stanu, mogą wyłącznie wywołać inne metody lub przetworzyć dane wejściowe.

Przykład

```
interface Intermediary {
    class Data{
        String name;
        String value;
    }
    Data readFromSource(Object source);
    boolean hasNextData(Object source);
    boolean writeToTarget(Object target, Data d);
    default boolean transmit(Object source, Object target){
        while(hasNextData(source)){
            Data d = readFromSource(source);
            if(!writeToTarget(target,d))return false;
        }
        return true;
    }
}
```

Jeszcze raz przykład IInnerClass

```
interface IInnerClass
{
 class A  {
 void foo(){System.out.println("A->foo()");}
 }
 // standardowa implementacja
 default void useA(A a){
   a.foo();
 }
}
class ITest implements IInnerClass{
  // odziedziczona implementacja useA(A a)
}

public class Test {
  public static void main(String[] args) {
    new ITest().useA(new IInnerClass.A());
    // A nawet (patrz następna część)
    new IInnerClass(){}.useA(new IInnerClass.A());
  }
}
```

Konflikt standardowych implementacji odziedziczonych różnymi drogami.

Na podstawie [http://stackoverflow.com/questions/18286235/what-is-the-default-implementation-of-method-defined-in-an-interface]

```
public interface A {
    default void foo(){
        System.out.println("Calling A.foo()");
    }
}

public interface B {
    default void foo(){
        System.out.println("Calling B.foo()");
    }
}


public class ClassAB implements A, B {

}
```

```
java: class ClassAB inherits unrelated defaults for foo() from
types A and B
```

Konieczna manualna implementacja

```
public class ClassAB implements A, B {
    public void foo(){}
}
```

Można w niej odwołać się do jednej ze standardowych implementacji

```
public class ClassAB implements A, B {
    public void foo(){
       A.super.foo(); // czyli mamy też B.super
    }
}
```

Metody statyczne w interfejsach (od Java 1.8)

W interfejsach można implementować metody statyczne (ale atrybutów statycznych niebędących stałymi jednak nie).

```
public interface IUtils {
    static double square(double x){
        return x*x;
    }
    static double factorial(int n){
        double r =1;
        for(int i=1;i<=n;i++)r*=i;
        return r;
    }
}
```

W wersji Java 1.9 dodano możliwość definiowania prywatnych metod. Nie są one dziedziczone, ani nie są częścią publicznego interfejsu. Mogą być wołane z metod standardowych i statycznych.

Podsumowując kombinacje modyfikatorów metod interfejsu dla Java 1.8/1.9

`public static` – poprawna

`public abstract` – poprawna

`public default` – poprawna

`private static` – poprawna

`private abstract` – błędna

`private default` – błędna

`private` – poprawna

Klasy wewnętrzne i klasy zagnieżdżone

Pojęciem klasy wewnętrznej (ang. *inner class*) określana jest klasa zadeklarowana wewnątrz innej klasy, metody lub bloku instrukcji.

Klasy zagnieżdżone (ang. *nested class*) to klasy zadeklarowane wewnątrz klasy lub interfejsu poprzedzone modyfikatorem `static`.

Przykład

```
class Outer
{
 class Inner1 {} // wewnątrz klasy

void f()
{
 class Inner2 {} // wewnątrz metody
}
void g()
{
 int x=0;
 {
 class Inner3 {} // wewnątrz bloku
 }
 }
 static class Nested {} // zagnieżdżone
}
```

Klasy wewnętrzne i zagnieżdżone stosuje się w celu:

- ukrywania implementacji
- poprawy efektywności organizacji kodu w przypadku relacji dziedziczenia i implementacji

Mechanizm klas zagnieżdżonych jest analogiczny do deklaracji typów wewnętrznych w języku C++. Klasy wewnętrzne są bardziej skomplikowane, ponieważ ich obiekty nie mogą istnieć bez obiektów klasy zewnętrznej.

Ukrywanie implementacji

Poniższy przykład pokazuje zastosowanie klasy zagnieżdżonej do ukrywania implementacji.

Załóżmy, że tworzymy oprogramowanie dokonujące operacji na macierzach liczb rzeczywistych. Od struktur danych implementujących macierze wymaga się, aby realizowały one interfejs `Matrix`.

```
interface Matrix
{
    double get(int row, int col);
    void set(int row, int col, double value);
}
```

Klasa `SparseMarix` posługuje *rzadką* implementacją macierzy. Macierze pojawiające się w opisie wielu problemów zawierają często 95% lub więcej elementów zerowych. W implementacji rzadkiej pamiętane są jedynie elementy niezerowe: para indeksów określająca wiersz i kolumnę oraz wartość elementu.

W tym celu tworzymy prywatną wewnętrzną strukturę danych: klasę `Element` o polach `row`, `col` oraz `value`.

Klasa `SparseMatrix` zawiera tablicę obiektów typu `Element` oraz ich licznik `count`. Tablica ta jest dynamicznie powiększana przy dodawaniu nowych elementów.

Kod klasy

```
class SparseMatrix implements Matrix
{
    private static class Element
    {
        int row=-1, col=-1;
        double value=0;
    }
    List<Element> elements = new ArrayList<>();

    public double get(int row, int col)
    {
        for(Element e:elements)
            if(e.row==row && e.col==col)
                return e.value;
        return 0;
    }

    public void set(int row, int col, double val)
    {
        for(Element e:elements){
            if(e.row==row && e.col==col){
                e.value = val;
                return;
            }
        }
        if(val==0)return;
        Element ne = new Element();
        ne.row = row;ne.col=col;ne.value=val;
        elements.add(ne);
    }

}
```

Kod testowy

```
public class SparseMatrixTest
{
    static final int ROWS=10;
    static final int COLS=10;
    static void setUnityMatrix(Matrix m)
    {
        for(int i=0;i<ROWS;i++){
            for(int j=0;j<COLS;j++){
                m.set(i,j,i==j?1:0);
            }
        }
    }
    static void dump(Matrix m)
    {
        for(int i=0;i<ROWS;i++){
            for(int j=0;j<COLS;j++){
                System.out.print(m.get(i,j)+" ");
            }
            System.out.println();
        }
    }
    public static void main(String args[]){
        SparseMatrix m = new SparseMatrix();
        setUnityMatrix(m);
        dump(m);
    }
}
```

Wynik

```
0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

Metoda `setUnityMatrix(Matrix m)` inicjuje macierz o wartościach `1` na przekątnej.

Metoda `dump(Matrix m)` wypisuje zawartość macierzy.

Powyższy przykład pokazuje standardowe zastosowanie klas wewnętrznych do ukrywania implementacji. Utworzone struktury danych realizują ściśle określony interfejs programistyczny.

Utworzona biblioteka może zawierać szereg funkcji posługujących się tym interfejsem obejmujących inicjację macierzy, mnożenie, itp.

Interfejs nie narzuca ograniczeń na implementację; możliwa jest implementacja w postaci tablicy dwuwymiarowej elementów typu `double`, ale także przedstawiona implementacja rzadka (`SparseMatrix`).

W przypadku implementacji rzadkiej użyte wewnętrzne struktury danych mogą pozostać ukryte (klasa `Element` jest zadeklarowana jako `private`).

Relacje pomiędzy klasami zewnętrznymi i wewnętrznymi

Klasa wewnętrzna zadeklarowana jest w ciele klasy zewnętrznej bez użycia modyfikatora `static`.

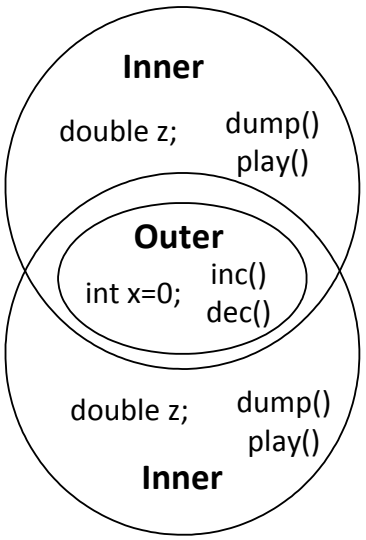
Potencjalnie oznacza to, że:

- Obiekt klasy zewnętrznej może istnieć bez obiektu klasy wewnętrznej.
- Obiekt klasy wewnętrznej nie może istnieć bez obiektu klasy zewnętrznej.
- Obiekt klasy wewnętrznej może odwoływać się do (niestatycznych) pól i metod obiektu klasy zewnętrznej.
- Może istnieć większa liczba obiektów klasy wewnętrznej „dzielących" obiekt klasy zewnętrznej.
- Z metod obiektu klasy wewnętrznej mamy dostęp do wszystkich (w tym prywatnych) pól i metod obiektu klasy zewnętrznej.

Przykład

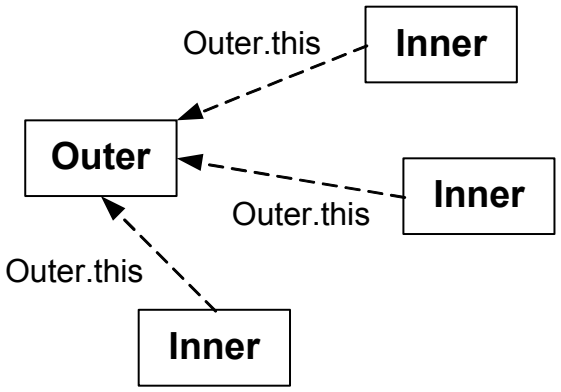
```
class Outer
{
 private int x=0;
 void inc(){x++;}
 void dec(){x--;}

 class Inner
 {
  double z = 0;
  void dump() {
  System.out.println(
    "Outer[x="+x+"];z="+z);
  }
  void play() {
     inc();inc();
     z=x;dec();
  }
 }
}
```



Pomiędzy klasami wewnętrznymi i zewnętrznymi nie zachodzi relacja kompozycji!

Obiekty klasy wewnętrznej `Inner` są połączone z obiektem klasy zewnętrznej `Outer` za pośrednictwem referencji: `Outer.this`.



Odwołanie do pól lub metod klasy zewnętrznej jest więc równoznaczne wywołaniu `Outer.this.`*`nazwapola`* lub `Outer.this.`*`nazwametody`*`().`

Metoda `play()` może zostać zaimplementowana jako:

```
void play()
{
   Outer.this.inc(); Outer.this.inc();
   z=Outer.this.x;Outer.this.dec();
}
```

Referencja `Outer.this` może być używana dla rozwiązywana szczególnych przypadków ukrywania metod i pól.

Przykład

```
class O
{
 void dump(){System.out.println("O[x="+x+"]");}
 int x=0;
 class Inner{
 double x=7;
 void dump(){
   System.out.println("Inner[x="+x+"]");
   O.this.dump();
   }
 }
}
```

Wywołanie metody `dump()` obiektu klasy `Inner` wypisze:

```
Inner[x=7.0] O[x=0]
```

Pomiędzy obiektem klasy zewnętrznej i wewnętrznej brak jest tego typu wbudowanej asocjacji. W typowym przypadku:

- obiekt klasy zewnętrznej tworzy obiekt klasy wewnętrznej i przechowuje referencję do niego w zmiennej lub pewnej kolekcji zmiennych;
- obiekt klasy zewnętrznej implementuje funkcję, która tworzy obiekt klasy wewnętrznej i zwraca referencję do niego na zewnątrz. Wówczas traci on dostęp do obiektu klasy wewnętrznej.

Tworzenie obiektów klasy wewnętrznej

Utworzenie obiektu klasy wewnętrznej może odbywać się wyłącznie w kontekście, w którym istnieje otaczająca instancja klasy zewnętrznej (ang. *enclosing instance*). Referencja do obiektu klasy zewnętrznej `Outer.this` musi być przekazana do obiektu klasy wewnętrznej. W przeciwnym przypadku kompilator będzie raportował błąd.

- Jeżeli konstrukcja obiektu klasy wewnętrznej odbywa się wewnątrz metody klasy zewnętrznej, wówczas otaczającą instancją jest obiekt, którego metodę wywołano.
- Jeżeli konstrukcja odbywa się na zewnątrz, wówczas operator `new` tworzący obiekt klasy wewnętrznej jest traktowany tak, jak metoda klasy zewnętrznej. Wymagane jest podanie referencji do obiektu: `outerClassRef.new InnerClass()`.

Przykład

```
class Outer
{
 class Inner{
    void dump(){
     System.out.println( "Outer.Inner.dump");
    }
   }
 Inner getInner(){return new Inner();}
}
```

Tworzenie obiektów

```
public static void main(String args[]){
  Outer o = new Outer();
  Outer.Inner i= o.getInner(); // w met.
  i.dump();
  i=o.new Inner(); // na zewnątrz
  i.dump();
  i = new Outer().new Inner();
  i.dump();
}
```

Konieczność dostarczenia otaczającej instancji obiektu zewnętrznego dotyczy także przypadku, kiedy dziedziczymy po klasie wewnętrznej. Wówczas obiekt klasy zewnętrznej musi zostać stworzony w konstruktorze lub przekazany jako parametr konstruktora.

```
class Outer
{
 int x;
 Outer(int _x){x=_x;}
 class Inner{
   void dump(){System.out.println("x="+x);}
 }
}

class ChildOfInner extends Outer.Inner
{
 ChildOfInner(int z){
   new Outer(z).super();
 }
}
```

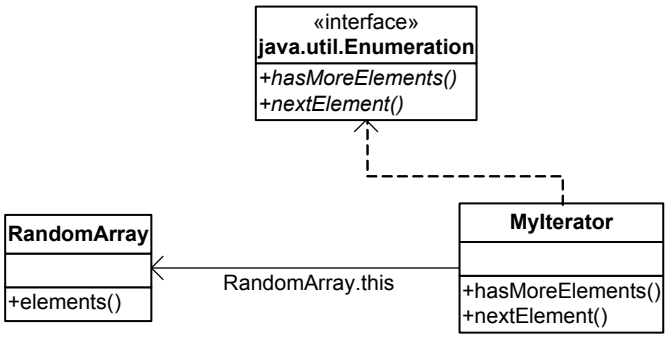
```
public static void main(String args[]){
  new ChildOfInner(7).dump();
}
```

Zastosowanie klas wewnętrznych

Mechanizm klas wewnętrznych umożliwia tworzenie obiektów, które

- dziedziczą po zewnętrznej klasie lub implementują zewnętrzny interfejs
- posiadają własny stan, niezależny od stanu innych obiektów tej samej klasy
- mogą odczytywać lub modyfikować stan klasy zewnętrznej

Przykład



Klasa `RandomArray` jest pewną kolekcją elementów. Jej implementacja ma być ukryta z zewnątrz.

W języku Java zdefiniowano standardowy interfejs iteracji po zawartości kolekcji: `java.util.Enumeration`.

- Metoda `boolean hasMoreElements()` zwraca informację, czy stan iteratora pozwala na odczyt kolejnego elementu.
- Metoda `Object nextElement()` zwraca referencję do bieżącego elementu i przesuwa kursor na następny element.

Sposób iteracji po zawartości kolekcji jest zazwyczaj ukryty. Zamiast tego kolekcje implementują metodę `Enumeration elements()`, która tworzy obiekt umożliwiający iterację po zawartości kolekcji implementujący interfejs `Enumeration`. Metody te zwracają referencję do obiektu dokonując wcześniej rzutowania w górę do referencji typu bazowego interfejsu.

Typowy fragment kodu umożliwiający sekwencyjny dostęp do elementów kolekcji ma postać:

```
Enumeration e = collection.elements();
while(e.hasMoreElements()){
   Object o = e.nextElement();
}
```

Implementacja

```
class RandomArray
{
  private int[]e;
  RandomArray(int size) {
  e=new int[size];
  java.util.Random r = new java.util.Random();
  for(int i=0;i<size;i++)e[i]=r.nextInt()%100;
  }

  private class MyIterator
  implements java.util.Enumeration
 {
    int i=0; // kursor
    public boolean hasMoreElements(){
      return i<e.length;
    }
    public Object nextElement(){
      return new Integer(e[i++]);
    }
  }
  java.util.Enumeration elements(){
    return new MyIterator();}
}
```

Kod testowy

```
public static void main(String args[])
{
  RandomArray r =new RandomArray(10);
  java.util.Enumeration e = r.elements();
  while(e.hasMoreElements())
    System.out.print(e.nextElement()+" ");
  System.out.println();
}
```

Lokalna klasa wewnętrzna

Jeżeli nie ma potrzeby, aby klasa wewnętrzna była jawnie dostępna na zewnątrz lub w kilku metodach klasy zewnętrznej – można zadeklarować ją w lokalnym kontekście, np.: metody lub bloku instrukcji.

Dla poprzedniego przykładu: definicja klasy `MyIterator` może zostać przeniesiona bezpośrednio do metody `elements()`.

Przykład

```
java.util.Enumeration elements()
{
  class MyIterator
  implements java.util.Enumeration
  {
     int i=0;
     public boolean hasMoreElements(){
      return i<e.length;}
     public Object nextElement(){
      return new Integer(e[i++]);}
   }

   return new MyIterator ();
}
```

Anonimowe klasy wewnętrzne

Konstrukcja bibliotek w języku Java stosunkowo często narzuca sytuację, kiedy powinniśmy utworzyć obiekt klasy implementującej określony interfejs lub dziedziczącej po pewnej klasie i następnie zwrócić do niego referencję lub przekazać ją do funkcji.

Wymagana klasa może być zdefiniowana w lokalnym kontekście i nie ma nawet potrzeby określania jej nazwy.

Przykładem może być implementacja metody `elements()`, która posługuje się lokalną anonimową klasą.

```
java.util.Enumeration elements(){
  return new java.util.Enumeration()
  {
   int i=0;
   public boolean hasMoreElements(){
    return i<e.length;}
   public Object nextElement(){
    return new Integer(e[i++]);}
  };
}
```

Metoda `elements()` zwraca referencję do klasy implementującej interfejs `java.util.Enumeration`. Pełna definicja klasy znajduje się w nawiasach klamrowych bezpośrednio po wywołaniu operatora `new`. Klasa nie ma jawnie określonej nazwy. Zostanie ona automatycznie wygenerowana przez kompilator.

Klasy anonimowe są bardzo szeroko stosowane przy implementacji graficznego interfejsu aplikacji.

W każdym środowisku graficznym akcja użytkownika, jak: naciśnięcie przycisku, naciśnięcie przycisku myszy, klawisza, wybranie elementu menu, itd. generuje zdarzenie. Dalej na postawie typu zdarzenia i jego parametrów sterowanie jest przekazywane do odpowiedniej funkcji (*callback function*).

W bibliotece AWT i Swing zdarzenia nie są kierowane bezpośrednio do funkcji, ale do obiektów implementujących grupę interfejsów: `ActionListener`, `KeyListener`, `MouseListener`, `MouseMotionListener`, itd.

Z punktu widzenia zarządzania kodem najbardziej efektywnym jest zaimplementowanie szeregu anonimowych klas wewnętrznych powiązanych z poszczególnymi elementami interfejsu użytkownika i zaimplementowanie w nich funkcji, które powinny być wywołane w odpowiedzi na pojawienie się zdarzenia.

Klasy zagnieżdżone

Klasy zagnieżdżone (ang. *nested classes*) to klasy zadeklarowane wewnątrz innych klas lub interfejsów z modyfikatorem `static`. W odróżnieniu od klas wewnętrznych, ich obiekty mogą istnieć samodzielnie, bez otaczającej instancji klasy zewnętrznej.

W metodach klas zagnieżdżonych nie jest dostępna referencja `Outer.this`.

- Nie jest możliwe wywołanie metod obiektu (niestatycznych) i dostęp do pól obiektu.
- Możliwy jest natomiast dostęp do (statycznych) pól i metod klasy.
- W odróżnieniu od klas wewnętrznych, klasy zagnieżdżone mogą definiować pola i metody statyczne. (Klasy wewnętrzne mogą definiować jedynie atrybuty typu `static final`, czyli stałe.

Przykład

```
class Outer {
 static int x=0;
 static void f(){System.out.println("Outer.f");}
 static class Nested{
   static int y=2;
   void dump(){System.out.println("x="+x);}
   void g(){f();}
 }
}
```

Nazwy wygenerowanych klasy

W języku Java każda wygenerowana klasa ma swój wewnętrzny identyfikator (używany między innymi przy RTTI – Run Time Type Identification). Identyfikatory te są równocześnie nazwami plików `.class` powstającymi w wyniku kompilacji.

Podczas tworzenia nazw przyjęto prostą zasadę: w przypadku klas wewnętrznych ich nazwy tworzy się przez połączenie nazwy klasy zewnętrznej i wewnętrznej znakiem $.

W przypadku klasy lokalnej po znaku $ dodawana jest kolejna liczba i nazwa klasy. Jeżeli klasa jest anonimowa używana jest wyłącznie liczba:

Outer.class	Nazwa pliku klasy zewnętrznej.
Outer$Inner.class	Nazwa pliku klasy wewnętrznej.
Outer$1LocalInner.class	Nazwa pliku klasy lokalnej wewnętrznej
Outer$1.class	Nazwa pliku klasy anonimowej

Podsumowanie

Klasy wewnętrzne stosuje się dla

- ukrywania wewnętrznej implementacji klasy
- jako element pozwalający dostosować architekturę klasy do wymagań określających jej miejsce w hierarchii dziedziczenia oraz realizowanych przez nią interfejsów.

W tym drugim przypadku klasa wewnętrzna najczęściej dziedziczy po innej klasie lub interfejsie, a równocześnie jej kod pozwala na pełny dostęp do zawartości obiektu klasy zewnętrznej.

Potencjalnie, zamiast implementować interfejs w klasie wewnętrznej, możemy zdecydować się na jego implementację w klasie zewnętrznej. Nie zawsze jest to jednak zgodne z zaleceniami projektowania oprogramowania.

- Na przykład implementacja interfejsu `Enumeration` przez `RandomArray` narzucałaby konieczność umieszczenia kursora w klasie. Nie byłaby możliwa równoczesna iteracja dokonywana przez współbieżne wątki.
- Przy konstrukcji interfejsu graficznego konieczne jest dostarczenie obiektów, które będą odbiorcami zdarzeń generowanych w wyniku akcji użytkownika. Tym obiektem może być główna klasa okna, ale taka implementacja jest bardziej podatna na przypadkowe błędy.

Przykład

```
class MyFrame extends Frame
implements ActionListener
{
void construct (Menu m)
{
    MenuItem item = new MenuItem("New");
    item.setActionListener(this);
    m.add(item);
    item = new MenuItem("Open");
    item.setActionListener(this);
    m.add(item);
    // itd
}
public void actionPerformed(ActionEvent e)
{
    String c = e.getActionCommand();
    if(c.equals("New")){/* reakcja na new */}
    if(c.equals("Open")){/* reakcja na open */}
    // itd
}
}
```

Taka implementacja jest dopuszczalna, ale nie jest zalecana w oficjalnej dokumentacji, ponieważ segregacja zdarzeń i uaktywniane kodu w reakcji na zdarzenie opiera się na porównaniu tekstów.

W zamian zaleca się tworzenie kodu, w którym do każdej akcji przypisany jest obiekt klasy wewnętrznej implementującej interfejs `ActionListener`.

We współczesnych środowiskach IDE tego typu kod powstaje i jest zarządzany automatycznie w procesie wizualnego programowania.

Przykład implementacji

```
void construct (Menu m)
{
MenuItem item = new MenuItem("New");
item.setActionListener( new ActionListener()
    {
        public void actionPerformed(ActionEvent e){
        // rekacja na polecenie New, np.:
        // onNew();
        }
    });
m.add(item);

item = new MenuItem("Open");
item.setActionListener( new ActionListener()
    {
        public void actionPerformed(ActionEvent e){
        // rekacja na polecenie Open
        }
    });
m.add(item);
// itd
}
```

- Użycie klas wewnętrznych może być jedynym rozwiązaniem problemu wielokrotnego dziedziczenia.

Przykład

```
abstract class A {
 abstract void f();
}

abstract class B {
 abstract void g();
}
```

```
class Outer extends A{
 void f(){}
 class Inner extends B {
   void g(){}
 }
}
```

- Można stworzyć wiele instancji klasy wewnętrznej powiązanych z obiektem klasy zewnętrznej. Obiekty klasy wewnętrznej przechowują swój stan, a także pozwalają na manipulację obiektem klasy zewnętrznej.

Przykładem może być równoczesna iteracja po zawartości klasy `RandomArray` (lub innej kolekcji) prowadzona np.: przez współbieżnie biegnące wątki.

- Obiekty klasy wewnętrznej mogą być tworzone na żądanie, niekoniecznie w momencie tworzenia obiektu klasy zewnętrznej.
- W początkowych fazach rozwoju języka Java krytykowano użycie klas wewnętrznych, zwłaszcza przy programowaniu apletów, ponieważ generowały dodatkowe pliki.

Ładowanie dużej liczby plików przez sieć jest nieefektywne, ponieważ za każdym razem otwierane jest połączenie, tworzony wątek serwera internetowego, zachodzi transfer danych, zamykanie połączenia i usuwanie wątku, itd. Dla małych plików narzut na inicjację i zamykanie połączenia jest duży w stosunku do czasu transferu.

Pierwsze aplety miały postać monolitycznej pojedynczej klasy bez jakichkolwiek struktur danych, które mogłyby wygenerować dodatkowe pliki *class*. Na przykład struktury danych klasy `SparseMatrix` byłyby w tym stylu raczej zaimplementowana jako:

```
class SparseMatrix
{
 int []rows = new int[100];
 int []cols = new int[100];
 double[] values = new int[100];
 int count=0;
 // itd
}
```

Problem ten należy rozwiązywać umieszczając wygenerowane pliki *class* w archiwum (*jar*, *zip* lub *cab*) i w ten sposób dystrybuując aplet lub aplikację. Maszyna wirtualna Java może uruchomić aplet lub aplikację pobierając kod klas ze wskazanego archiwum.

Typy wyliczeniowe

Typ wyliczeniowy (enum) pozwala na zdefiniowanie zbioru symboli.

```
public enum WeekDay {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Następnie można zadeklarować zmienne typu wyliczeniowego oraz wykorzystywać symbole do przypisania, porównania.

```
WeekDay day = WeekDay.FRIDAY;
switch (day){
    case MONDAY:
    case TUESDAY:
    case WEDNESDAY:
    case THURSDAY:
        break;
    case FRIDAY:
        System.out.println("Black friday");
    case SUNDAY:
    case SATURDAY:
        System.out.println("Weekend");
        break;
}
```

Typy wyliczeniowy `EnumType` jest w rzeczywistości klasą (dziedziczącą po `java.lang.Enum`), natomiast symbole wyliczeniowe są obiektami tej klasy.

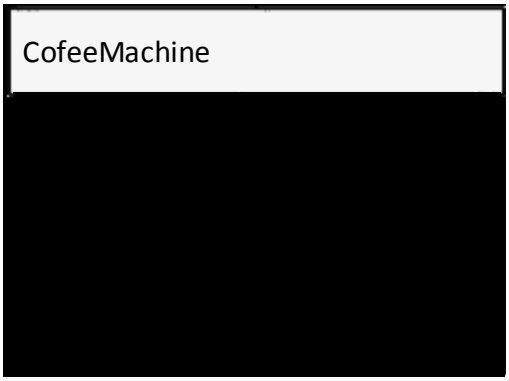
Metody

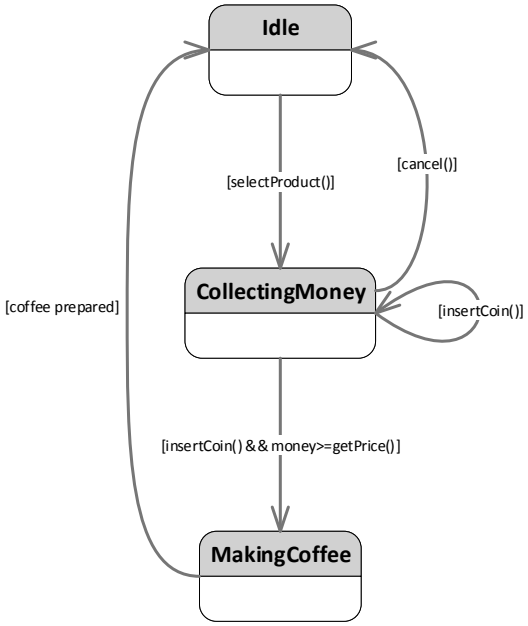
- Statyczna funkcja `EnumType.values()` zwraca tablicę zdefiniowanych symboli
- `String name()` zwraca nazwę symbolu
- `int ordinal()` zwraca numer porządkowy symbolu

Przykład

```
for(WeekDay w:WeekDay.values()){
    System.out.println(w.name()+":"+w.ordinal());
}
```

Przykład – maszyna stanowa






```
public class CoffeeMachine {
    enum State {Idle, CollectingMoney, MakingCoffee};
    State state = State.Idle;
    int product=-1;
    double money=0;

    double getPrice(int product){
        return 2.50;
    }
    void selectProduct(int id){
        if(state!=State.Idle)return;
        product = id;
        money = 0;
        state = State.CollectingMoney;
    }
```

```
    void insertCoin(double coin){
        if(state!=State.CollectingMoney){
            System.out.printf("Returning %f coin\n",coin);
        }else{
            money+=coin;
            double rest = money - getPrice(product);
            if(rest<0)return;
            if(rest>0){
                System.out.printf("Returning %f rest\n",rest);
            }
            state = State.MakingCoffee;
            makeCoffee();
        }
    }

    void cancel(){
        if(state==State.CollectingMoney){
            System.out.printf("Returning %f money\n",money);
            money=0;
            state=State.Idle;
        }
    }
    void makeCoffee(){
        System.out.println("Making caffee");
        state=State.Idle;
    }
}
```

Typy wyliczeniowe mogą mieć atrybuty, konstruktory i metody dostępu.

```
public enum Directions {
    NORTH(0,1),
    SOUTH(0,-1),
    EAST(1,0),
    WEST(-1,0);

    private double x;
    private double y;
    Directions(double x, double y){
        this.setX(x);
        this.setY(y);
    }
    public String toString(){
        return name()+"("+ getX() +", "+ getY() +")";
    }

    public double getX() {
        return x;
    }
    public void setX(double x) {
        this.x = x;
    }
    public double getY() {
        return y;
    }
    public void setY(double y) {
        this.y = y;
    }
}
```

```
for(Directions d : Directions.values()){
    System.out.println(d.toString());
}
```

Wynik

```
NORTH(0.0, 1.0)
SOUTH(0.0, -1.0)
EAST(1.0, 0.0)
WEST(-1.0, 0.0)
```

Zastosowanie setterów jest dyskusyjne…

Typy wyliczeniowe mogą również implementować interfejsy. Każdy z obiektów składowych może wówczas przedefiniować lokalnie funkcje zdefiniowane w interfejsie.

```
public interface Evaluable {
    double eval(double x);
}
```

```
public enum Functions implements Evaluable{
    SIN {
        public double eval(double x){
            return Math.sin(x);
        }
    },
    X2SIN {
        public double eval(double x) {
            return x * x * Math.sin(x);
        }
    },
    SIGMOID {
        public double eval(double x) {
            return 1/( 1 + Math.exp(-x));
        }
    };
}
```

```
for(Functions f:Functions.values()){
    System.out.printf(Locale.US,
   "%s(%f)=%f\n",f.name(),1.2,f.eval(1.2));
}
```

Wynik

```
SIN(1.200000)=0.932039
X2SIN(1.200000)=1.342136
SIGMOID(1.200000)=0.768525
```