

JAVA

Assertions

Assertion

- od JDK 1.4
- příkaz obsahující výraz typu boolean
- programátor předpokládá, že výraz bude vždy splněn (**true**)
- pokud je výraz vyhodnocen na **false** -> chyba
- používá se pro ladění
 - assertions lze zapnout nebo vypnout
 - pro celý program nebo jen pro některé třídy
 - implicitně vypnuty
 - **nesmí** mít žádné vedlejší efekty
- JDK 1.4 – programy se musí překládat s parametrem
-source 1.4

Použití

```
assert Vyraz1;  
assert Vyraz1 : Vyraz2;
```

- vypnuté assertions – příkaz nedělá nic
 - výrazy se nevyhodnocují!
- zapnuté assertions
 - Výraz1 je true – nic se neděje, program pokračuje normálně
 - Výraz1 je false
 - Výraz2 je přítomen
`throw new AssertionError(Vyraz2)`
 - Výraz2 není přítomen
`throw new AssertionError()`

Zapnutí a vypnutí

- parametry pro virtual machine
- zapnutí
 - ea[:PackageName...]:ClassName]
 - enableassertions[:PackageName...]:ClassName]
- vypnutí
 - da[:PackageName...]:ClassName]
 - disableassertions[:PackageName...]:ClassName]
- bez třídy nebo balíku – pro všechny třídy
- assertions v "systémových" třídách
 - esa | -enablesystemasserions
 - dsa | -disablesystemasserions
- zda se mají assetions provádět, se určí pouze jednou při inicializaci třídy (předtím, než se na ní cokoliv provede)

java.lang.AssertionError

- dědí od `java.lang.Error`
- konstruktory
 - `AssertionError()`
 - `AssertionError(boolean b)`
 - `AssertionError(char c)`
 - `AssertionError(double d)`
 - `AssertionError(float f)`
 - `AssertionError(int i)`
 - `AssertionError(long l)`
 - `AssertionError(Object o)`

Příklady použití

- invarianty

```
if (i%3 == 0) {  
    ...  
} else if (i%3 == 1) {  
    ...  
} else {  
    assert i%3 == 2;  
    ...  
}
```

Příklady použití

- "nedosažitelná místa" v programu

```
class Directions {
    public static final int RIGHT = 1;
    public static final int LEFT = 2;
}
...
switch(direction) {
    case Directions.LEFT:
        ...
    case Directions.RIGHT:
        ...
    default:
        assert false;
}
```

Příklady použití

- preconditions

- testování parametrů `private` metod

```
private void setInterval(int i) {  
    assert i>0 && i<=MAX_INTERVAL;  
    ...  
}
```

- nevhodné na testování parametrů `public` metod

```
public void setInterval(int i) {  
    if (i<=0 && i>MAX_INTERVAL)  
        throw new IllegalArgumentException();  
    ...  
}
```

Příklady použití

- postconditions

```
public String foo() {  
    String ret;  
    ...  
    assert ret != null;  
    return ret;  
}
```

Generické typy

Úvod

- *obdoba* šablon z C#/C++
 - **ale jen na první pohled**
- parametry pro typy
- cíl
 - přehlednější kód
 - typová bezpečnost

Motivační příklad

- bez gen. typů (<=JDK 1.4)

```
List myIntList = new LinkedList();  
myIntList.add(new Integer(0));  
Integer x = (Integer)myIntList.iterator().next();
```

- JDK 5.0

```
List<Integer> myIntList = new LinkedList<Integer>();  
myIntList.add(new Integer(0));  
Integer x = myIntList.iterator().next();
```

- bez explicitního přetypování
- kontrola typů během překlada

Definice generických typů

```
public interface List<E> {  
    void add(E x);  
    Iterator<E> iterator();  
    E get(int i);  
}  
  
public interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}
```

- `List<Integer>` si lze představit jako

```
public interface IntegerList {  
    void add(Integer x);  
    Iterator<Integer> iterator();  
}
```

- ve skutečnosti ale takový kód nikde neexistuje
– negeneruje se kód jako C++

Vztahy mezi typy

- nejsou povoleny žádné změny v typových parametrech

```
List<String> ls = new ArrayList<String> ();  
List<Object> lo = ls;
```

```
lo.add(new Object());  
String s = ls.get(0);
```

chyba – přiřazení Object do String

- druhý řádek způsobí chybu při překladu

Vztahy mezi typy

- příklad - tisk všech prvků kolekci
≤ JDK 1.4

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++) {  
        System.out.println(i.next());  
    }  
}
```

naivní pokus v JDK 5.0

```
void printCollection(Collection<Object> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

- nefunguje (viz předchozí příklad)

Vztahy mezi typy

- `Collection<Object>` není nadtyp všech kolekcí
- **správně**

```
void printCollection(Collection<?> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```
- `Collection<?>` je nadtyp všech kolekcí
 - kolekce neznámého typu (collection of unknown)
 - lze přiřadit kolekci jakéhokoliv typu
- **pozor** - do `Collection<?>` nelze přidávat
 - `Collection<?> c = new ArrayList<String>();`
 - `c.add(new Object());` **<= chyba při překladu**
- **volat** `get()` lze - výsledek do typu `Object`

Vztahy mezi typy

- ? - wildcard
- „omezený ?“ (bounded wildcard)

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}
public class Circle extends Shape { ... }
public class Canvas {
    public void drawAll(List<Shape> shapes) {
        for (Shape s:shapes) {
            s.draw(this)
        }
    }
}
```

- umožní vykreslit pouze seznamy přesně typu `List<Shape>`, ale už ne `List<Circle>`

Vztahy mezi typy

- řešení - omezený ?

```
public void drawAll(List<? extends Shape> shapes) {  
    for (Shape s:shapes) {  
        s.draw(this)  
    }  
}
```

- do tohoto Listu stále nelze přidávat

```
shapes.add(0, new Rectangle()); chyba při překladu
```

Generické metody

```
static void fromArrayToCollection(Object[] a,  
    Collection<?> c) {  
    for (Object o : a) {  
        c.add(o); ← chyba při překladu  
    }  
}
```

```
static <T> void fromArrayToCollection(T[] a,  
    Collection<T> c) {  
    for (T o : a) {  
        c.add(o); ← OK  
    }  
}
```

Generické metody

- použití
 - překladač sám určí typy

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
toArray(oa, co); // T → Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
toArray(sa, cs); // T → String
toArray(sa, co); // T → Object
```

- i u metod lze použít omezený typ

```
class Collections {
    public static <T> void copy(List<T> dest, List<?
    extends T> src) {...}
}
```

Pole a generické typy

- pole gen. typů
 - lze deklarovat
 - nelze naalokovat

```
List<String>[] lsa = new List<String>[10]; nelze!  
List<?>[] lsa = new List<?>[10]; OK + varování
```

- proč - pole lze přetypovat na Object

```
List<String>[] lsa = new List<String>[10];  
Object o = lsa;  
Object[] oa = (Object[]) o;  
List<Integer> li = new ArrayList<Integer>();  
li.add(new Integer(3));  
oa[1] = li;  
String s = lsa[1].get(0); ClassCastException
```

„Starý“ a „nový“ kód

- „starý“ kód bez generických typů

```
public class Foo {  
    public void add(List lst) { ... }  
    public List get() { ... }  
}
```

- „nový“ kód používající „starý“

```
List<String> lst1 = new ArrayList<String>();  
Foo o = new Foo();  
o.add(lst1); ← OK - List odpovídá List<?>  
List<String> lst2 = o.get(); ← varování překladače
```

„Starý“ a „nový“ kód

- „nový“ kód s generickými typy

```
public class Foo {  
    public void add(List<String> lst) { ... }  
    public List<String> get() { ... }  
}
```

- „starý“ kód používající „nový“

```
List lst1 = new ArrayList();  
Foo o = new Foo();  
o.add(lst1); ← varování překladače  
List lst2 = o.get(); ← OK - List odpovídá List<?>
```

Další vztahy mezi typy

```
class Collections {  
    public static <T> void copy(List<T> dest, List<?  
    extends T> src) {...}  
}
```

- ve skutečnosti

```
class Collections {  
    public static <T> void copy(List<? super T> dest,  
    List<? extends T> src) {...}  
}
```

Enum

Výčty

- **<= JDK 1.4**

```
public static final int COLOR_BLUE = 0;  
public static final int COLOR_RED = 1;  
public static final int COLOR_GREEN = 2;
```

- **možné problémy**

- typová (ne)bezpečnost
- žádný namespace
- konstanty napevno přeložené v klientech
- při výpisu jen hodnoty

Enum

```
public enum Color { BLUE, RED, GREEN }  
...  
public Color clr = Color.BLUE;
```

- „normalní“ třída
 - atributy, metody, i metodu main
 - potomek třídy `java.lang.Enum`
 - pro každou konstantu - jedna instance
 - `public static final` atribut
 - `protected` konstruktor

java.lang.Enum

```
public abstract class Enum <E> extends  
    Enum<E>> { ... }
```

- metody
 - String name()
 - int ordinal()
- každý enum má metodu values()
 - vrací pole se všemi konstantami

```
public Colors clr = Colors.BLUE;  
System.out.println(clr);    → BLUE
```

Attributy a metody

```
public enum Planet {  
    MERCURY (3.303e+23, 2.4397e6),  
    VENUS (4.869e+24, 6.0518e6),  
    EARTH (5.976e+24, 6.37814e6),  
    ...  
  
    private final double mass;  
    private final double radius;  
  
    Planet(double mass, double radius) {  
        this.mass = mass;  
        this.radius = radius;  
    }  
  
    double surfaceGravity() {  
        return G * mass / (radius * radius);  
    }  
}
```

Atributy a metody

- příklad

```
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    double eval(double x, double y) {
        switch(this) {
            case PLUS:    return x + y;
            case MINUS:   return x - y;
            case TIMES:   return x * y;
            case DIVIDE:  return x / y;
        }
        throw new AssertionError("Unknown op: " + this);
    }
}
```

Atributy a metody

- abstraktní metody
- konkrétní implementace u každé konstanty

```
public enum Operation {  
    PLUS { double eval(double x, double y) { return x+y; }},  
    MINUS { double eval(double x, double y) { return x-y; }},  
    TIMES { double eval(double x, double y) { return x*y; }},  
    DIVIDE { double eval(double x, double y) { return x/y;}};  
  
    abstract double eval(double x, double y);  
}
```

Proměnný počet parametrů



- „tři tečky“
- pouze jako poslední parametr
- lze předat pole nebo seznam parametrů
- v metodě dostupné jako pole

```
void argtest(Object... args) {  
    for (int i=0;i <args.length; i++) {  
        System.out.println(args[i]);  
    }  
}  
  
argtest("Ahoj", "jak", "se", "vede");  
argtest(new Object[] {"Ahoj", "jak", "se",  
    "vede"});
```

- metoda printf
 - System.out.printf("%s %d\n", user, total);

Příklad

- Jsou volání ekvivalentní?

```
argtest("Ahoj", "jak", "se", "vede");  
argtest(new Object[] {"Ahoj", "jak", "se", "vede"});  
argtest((Object) new Object[] {"Ahoj", "jak", "se",  
    "vede"});
```

- a) Všechna ekvivalentní
- b) Ekvivalentní 1. a 2.
- c) Ekvivalentní 2. a 3.
- d) Každé dělá něco jiného