JAVA

Zdrojové soubory

Unicode

- programy ~ Unicode
  - komentáře, identifikátory, znakové a řetězcové konstanty
  - ostatní ~ ASCII (<128)
    - nebo Unicode escape sekvence < 128
- Unicode escape sekvence
  - `\uxxxx`
  - `\u0041 .... A`
- rozvinutá sekvence se už nepoužije pro další sekvence
  - `\u005cu005a`
    - šest znaků `\ u 0 0 5 a`

Postup zpracování programu

1. překlad unicode escape sekvencí (a celého programu) do posloupnosti unicode znaků
2. posloupnost z bodu 1 do posloupnosti znaků a ukončovačů řádků
3. posloupnost z bodu 2 do posloupnosti vstupních elementů (bez "bílých znaků" a komentářů)

ukončovače řádků

- CR LF
- CR
- LF

Test

```
public class Test {
  public static void main(String[] argv) {
    int i = 1;
    i += 1; // to same jako \u000A i = i + 1;
    System.out.println(i);
  }
}
```

- Program vypíše:
  a) 1
  b) 2
  c) 3
  d) nepřeloží se
  e) runtime exception

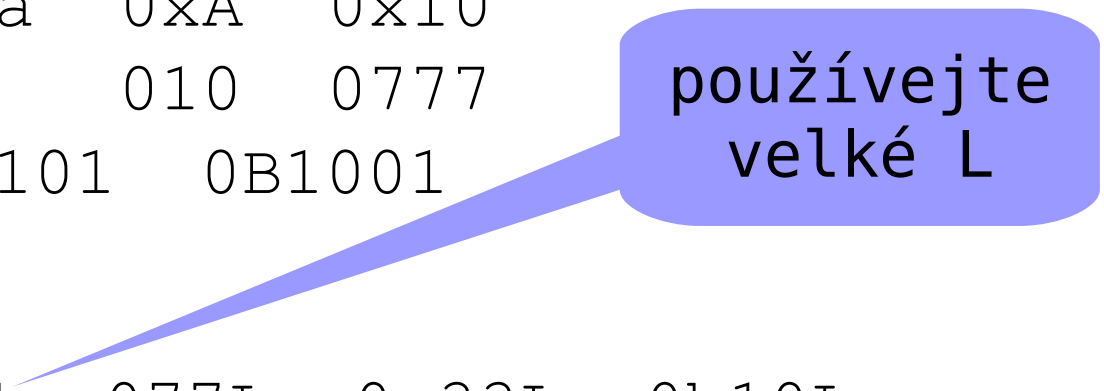
Encoding

- parametr pro javac `-encoding`
  - kódování zdrojových souborů
  - bez parametru – defaultní kódování

- v IDE – typicky vlastnost projektu

Literály

- integer literály
  - desítkové ... `0  1  23  -3`
  - šestnáctkové ... `0xa  0xA  0x10`
  - osmičkové ... `03  010  0777`
  - binární ... `0b101  0B1001`
    - od Java 7
  - implicitně typu int
  - typ long ... `1L  33l  077L  0x33L  0b10L`
- floating-point literály
  - `0.0  2.34  1.  .4  1e4  3.2e-4`
  - implicitně typu double
  - typ float ... `2.34f  1.F  .4f  1e4F  3.2e-4f`
- boolean literály
  - `true, false`



Literály

- podtržítka v numerických literálech
  - od Java 7
  - zlepšení čitelnosti

```
1234_5678_9012_3456L
999_99_9999L
3.14_15F
0xFF_EC_DE_5E
0xCAFE_BABE
0x7fff_ffff_ffff_ffffL
0b0010_0101
0b11010010_01101001_10010100_10010010
```

Literály

- znakové literály
  - `'a'  '%'  '\\'  '\''  '\u0045'  '\123'`
  - escape sekvence

`\b`	`\u0008`	back space
`\t`	`\u0009`	tab
`\n`	`\u000A`	line feed
`\f`	`\u000C`	form feed
`\r`	`\u000D`	carriage return
`\"`	`\u0022`	
`\'`	`\u0027`	
`\\`	`\u005c`	

Literály

- řetězcové literály
  - `""  "\""  "tohle je retezec"`
- víceřádkové řetězcové literály – od Java 13
  - preview feature – nutno povolit (--enable-preview)
  - `"""multiline`
    `string"""`
  - úvodní mezery (indentace) jsou odstraněny
    - přebytečné mezery na konci řádku také
  - `String html = """`

```
.............<html>
.............    <body>
.............        <p>Hello</p>
.............    </body>
.............</html>
.............""";
```

Literály

- `null` literál

Identifikátory

- identifikátor
  - jméno třídy, metody, atributu,...

- povolené znaky
  - číslice a písmena
    - nesmí začínat číslicí
  - ze speciálních znaků pouze _ a $
    - samotné podtržítko není povoleno
      - od Java 9

Identifikátory

- pojmenovávání
  - balíčky – všechna písmena malá
    - `cz.cuni.mff.java`
  - třídy, interfacy – `ListArray, InputStreamReader`
    - složeniny slov
    - „mixed case“
    - první písmeno velké
  - metody, atributy – `getSize, setColor`
    - složeniny slov
    - „mixed case“
    - první písmeno malé
  - konstanty – `MAX_SIZE`
    - vše velké
    - skládání přes podtržítko

Assertions

Assertion

- od Java 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

Použití

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

- vypnuté assertions – příkaz nedělá nic
  - výrazy se nevyhodnocují!
- zapnuté assetions
  - 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;
}
```

Java

Generické typy

Úvod

- od Java 5
- podobné generickým typům v C#
- parametry pro typy
- cíl
  - přehlednější kód
  - typová bezpečnost

Motivační příklad

- bez gen. typů (<=Java 1.4)

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

- >= Java 5

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

  - bez explicitního přetypování
  - kontrola typů během překladu

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

Překlad gen. typů

- zjednodušeně – při překladu se vymažou všechny informace o generických typech
  - „erasure“

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

- při běhu se kód chová jako

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

Překlad gen. typů

- stále stejná třída, i když je parametrizována čímkoliv
  - **`LinkedList<String>`**
  - **`LinkedList<Integer>`**
  - **`LinkedList<Foo>`**
  - …
- pouze jeden byte-code

- **nelze parametrizovat primitivními typy**
  - ~~List<int>~~

Vytváření objektů

```
ArrayList<Integer> list = new ArrayList<Integer>();
ArrayList<ArrayList<Integer>> list2 =
new ArrayList<ArrayList<Integer>>();
HashMap<String, ArrayList<ArrayList<Integer>>> h =
new HashMap<String, ArrayList<ArrayList<Integer>>>();
```

- od Java 7 (operátor „diamant“)

```
ArrayList<Integer> list = new ArrayList<>();
ArrayList<ArrayList<Integer>> list2 =
new ArrayList<>();
HashMap<String, ArrayList<ArrayList<Integer>>> h =
new HashMap<>();
```

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 řadek způsobí chybu při překladu

Vztahy mezi typy

- příklad - tisk všech prvků kolekci

  <= Java 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 Java 5

```
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>();
fromArrayToCollection(oa, co); // T → Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs); // T → String
fromArrayToCollection(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];
List<?>[] lsa = new List<?>[10];
```

nelze!

OK + varování

- proč - pole lze přetypovat na Object

```
List<String>[] lsa = new List<String>[10];
Object[] oa = (Object[]) lsa;
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){...}
}
```