

JAVA

Source files

Unicode

- programs ~ Unicode
 - comments, identifiers, char and string constants
 - the rest is in ASCII (<128)
 - or Unicode escape sequences < 128
- Unicode escape sequences
 - `\uxxxx`
 - `\u0041` `A`
- the expanded sequence is not used for following ones
 - `\u005cu005a` results in six chars
 - `\ u 0 0 5 a`

Source code file processing

1. translation of unicode escape sequences (and all of the source code) into a sequence of unicode chars
 2. the sequence from (1) is translated into a sequence of chars and line-terminating chars
 3. the sequence from (2) is translated into a sequence of input tokens (without white-spaces and comments)
- line-terminating chars
 - CR LF
 - CR
 - LF

Test

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

- Program prints out:
 - a) 1
 - b) 2
 - c) 3
 - d) cannot be compiled
 - e) a runtime exception

Encoding

- argument of javac `-encoding`
 - encoding of source files
 - without it – default encoding
- in IDE – typically a project property

Literals

- integer literals

- decimal ... 0 1 23 -3
- hexadecimal ... 0xa 0xA 0x10
- octal ... 03 010 0777
- binary ... 0b101 0B1001

use
capital L

- since Java 7

- by default of the **int** type

- **long** ... 1L 33L 077L 0x33L 0b10L

- floating-point literals

- 0.0 2.34 1. .4 1e4 3.2e-4

- by default **double**

- **float** ... 2.34f 1.F .4f 1e4F 3.2e-4f

- boolean literals

- true, false

Literals

- underscores in numerical literals
 - since Java 7
 - for better readability

```
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
```

Literals

- char literals

- 'a' ' % ' '\\ ' '\\ ' '\\u0045' '\\123'

- escape sequences

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

Literals

- String literals
 - `"\"` `"\"` `"this is a String"`
- multi-line string literals – since Java 13
 - preview feature – has to be enabled (`--enable-preview`)
 - `"""multiline
string"""`
 - initial spaces (indentation) are removed
 - trailing spaces too
 - `String html = """
.....<html>
.....<body>
.....<p>Hello</p>
.....</body>
.....</html>
.....""";`

Literals

- `null` literal

Identifiers

- identifier
 - name of class, method, field,...
- allowed characters
 - letters and digits
 - digit cannot be first character
 - special characters only `_` and `$`
 - standalone underscore is not allowed
 - since Java 9

Identifiers

- naming
 - packages – lowercase letters
 - `cz.cuni.mff.java`
 - class, interface – `ListArray`, `InputStreamReader`
 - composed words
 - mixed case
 - first letter capital
 - methods, fields – `getSize`, `setColor`
 - composed words
 - mixed case
 - first letter lower case
 - constants – `MAX_SIZE`
 - all letters upper case
 - composing via underscore

JAVA

Assertions

Assertion

- since Java 1.4
- the statement with a **boolean** expression
- a developer supposes that the expression is always satisfied (evaluates to **true**)
- if it is evaluated to **false** -> error
- intended for debugging
 - assertions can be enabled or disabled
 - for whole program or for several classes only
 - disabled by default
 - ***must not*** have any side effects

Usage

```
assert Expression1;
```

```
assert Expression1 : Expression2;
```

- disabled assertions – the statement does nothing
 - expressions are not evaluated
- enabled assertions
 - Expression1 is **true** – program continues normally
 - Expression1 is **false**
 - Expression2 is presented
`throw new AssertionError(Expression2)`
 - Expression2 is not presented
`throw new AssertionError()`

Enabling and disabling

- arguments for the virtual machine
- enabling
 - ea[:PackageName...]:ClassName]
 - enableassertions[:PackageName...]:ClassName]
- disabling
 - da[:PackageName...]:ClassName]
 - disableassertions[:PackageName...]:ClassName]
- without class or package – for all classes
- assertions in "system" classes
 - esa | -enablesystemassertions
 - dsa | -disablesystemassertions
- decision whether the assertions are enabled, is evaluated just once during initialization of a class (before anything is called/used on this class)

java.lang.AssertionError

- **extends** java.lang.Error
- **constructors**
 - AssertionError()
 - AssertionError(boolean b)
 - AssertionError(char c)
 - AssertionError(double d)
 - AssertionError(float f)
 - AssertionError(int i)
 - AssertionError(long l)
 - AssertionError(Object o)

Examples

- invariants

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

Examples

- "unreachable places" in a program

```
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;
}
```

Examples

- preconditions
 - testing arguments of `private` methods

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

- unrecommended for testing arguments of public methods

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

Examples

- postconditions

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

Java

Generics

Introduction

- since Java 5
- similar to the generics in C#
- typed arguments
- goal
 - clear code
 - type safety

Motivational example

- without generics (\leq Java 1.4)

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

- \geq Java 5

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

- no explicit casting
- type checks during compilation

Definition of generics

```
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>` can be seen as

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

- but in reality no such code exists

Compilation of gen. types

- to simplify – during compilation, all information about generic types are erased
 - "erasure"

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

- at runtime, it behaves as

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

Compilation of gen. types

- always the same class, even if parametrized by anything
 - `LinkedList<String>`
 - `LinkedList<Integer>`
 - `LinkedList<Foo>`
 - ...
- just a single byte-code
- **primitive types cannot be used as type parameters**
 - ~~`List<int>`~~

New instances

```
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>>>();
```

- since Java 7 (“diamond” operator)

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

Type relations

- no changes in typed arguments are allowed

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

```
lo.add(new Object());  
String s = ls.get(0);  
error – assigning Object to String
```

- second line causes compilation error

Type relations

- example – printing all elements in a collection
≤ Java 1.4

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

naive attempt in Java 5

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

- does not work (see the previous example)

Type relations

- `Collection<Object>` is not supertype of all collections

- correctly

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

- `Collection<?>` is supertype of all collections
 - collection of unknown
 - any collection can be assigned there
- **BUT** – to `Collection<?>` nothing can be added

```
Collection<?> c = new ArrayList<String>();  
c.add(new Object());
```

<= compilation error
- `get()` can be called – return type is `Object`

Type relations

- ? - wildcard
- 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)
        }
    }
}
```

- can draw lists of the type `List<Shape>` only but not e.g. `List<Circle>`

Type relations

- solution – bounded ?

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

- but still you cannot add to this List

```
shapes.add(0, new Rectangle()); compilation error
```

Generic methods

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

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

Generic methods

- usage
 - the compiler determines actual types automatically

```
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
```

- bounds can be used with methods also

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

Array and generics

- array of generics
 - can be declared
 - cannot be instantiated

```
List<String>[] lsa = new List<String>[10]; wrong  
List<?>[] lsa = new List<?>[10]; OK + warning
```

- why? arrays can be cast to 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
```

“Old” and “new” code

- “old” code without generics

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

- “new” code that uses the “old” one

```
List<String> lst1 = new ArrayList<String>();  
Foo o = new Foo();  
o.add(lst1); ← OK - List corresponds to List<?>  
List<String> lst2 = o.get(); ← compilation warning
```

“Old” and “new” code

- “new” code with generics

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

- “old” code that uses the “new” one

```
List lst1 = new ArrayList();  
Foo o = new Foo();  
o.add(lst1); ← compilation warning  
List lst2 = o.get(); ← OK - List corresponds to List<?>
```

Additional type relations

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

- actual declaration is

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



Slides version J04.en.2019.01

This slides are licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).