Object Oriented Programming

PART I

Introduction to Java

Beniamin Bogosel

Aurel Vlaicu University, Arad

Contact

Beniamin BOGOSEL: beniamin.bogosel@polytechnique.edu

Site Web: http://www.cmap.polytechnique.fr/~beniamin.bogosel/OOP_UAV.html

- slides
- lab subjects
- codes: Java

Course objectives

- Introduction to Java: Ecclipse programming environment
- Java syntax
- Object Oriented Programming: basic principles, applications

Why Java?

- programming language which **enforces OOP**
- used in industrial applications
- once OOP principles are understood, it is easy to switch between languages

Organization

Cours+Lab: Thursday 8:00-12:00 (with breaks)

⋆ switch back and forth between course and code to better understand the concepts underlined

⋆ **Strongly recommended**: to install Java and Ecclipse on your computer and try to code the examples yourselves

Initiate Java project Ecclipse: "Initiere Project Java.pdf"

1. Basic Java Syntax

Comments

```
/*
* Simple comment
* multiple lines
*/
// One end-line comment
/**
* Documentation comment
* (javadoc).
*/
```

Identifyers

- Construction: letters, digits, `$`, `_`
- Case sensitive `a`$\neq$`A`
- Unlimited length

Primitive types

⋆ 8 primitive types to work with

Type	Values	Size (bits)
`boolean`	false, true	1
`char`		16
`byte`		8
`short`	$-32768 \longrightarrow 32767$	16
`int`	$-2147483648 \longrightarrow 2147483647$	32
`long`	$-2^{63} \longrightarrow (2^{64} - 1)$	64
`float`	`1.4E-45` $\longrightarrow$ `3.4028235E38`	32
`double`	`4.9E-324` $\longrightarrow$ `1.7976931348623157E308`	64

Char type

⋆ Unicode characters: 16 unsigned bits
- Normal characters: `'A'`, other representations
- Special characters:
  - `'\b'` (backspace)
  - `'\t'` (tab)
  - `'\n'` (newline)
  - `'\f'` (newpage)
  - `'\r'` (carriage return)
  - `'\"'` (double quotes)
  - `'\''` (single quote)
  - `'\\'` (backslash)

Integer types

- base 10, base 8 or base 16
- `65` is a literal of type `int`
- `65L` is a literal of type `long`
- careful with computations made with `int` or `long`. try:

```
byte b = 1;
b = b + b;
```

vs

```
byte b = 1;
b = (byte) b + b;
```

- `b+b` is of type `int`.

Float types

- `float`: single precision, roughly 7 significant digits
- `3.14f`: literal of `float` type
- `double`: double precision, roughly 16 significant digits
- `3.14` or `3.14d` represent literals of type `double`

Conversion: loss of precision, loss of magnitude

```
float x = 123456795
```

...

```
byte x = (byte)1234567
```

⋆ converting integers to "bigger" integer: no loss of precision

⋆ integer to float: possible loss of precision

⋆ no error reporting in case problems happen.

⋆ converting to "smaller" type: possible precision loss, magnitude loss, no error reporting.

Strings

- ⋆ it is an object type
- ⋆ Literal between double quotes: `"BlablA Blabl\101 Blabl\u0041"`
- ⋆ No newline: use `\n` if necesssary

Assignment expressions

⋆ `v=e`
⋆ `v` is a variable identifyer
⋆ `e` is a compatible expression

- Simple assignemnt = expression with side effect having the value of `e`
  - side effect = modification of the value of `v`
  - expresion $\neq$ instruction
- Multiple assignments: `i = j = k = 2`
- Other assignemtn expressions:
  - `*=`, `+=`, `-=`, `/=`, ...
  - `v += e` $\Leftrightarrow$ `v = v + e`

Examples of expressions

`!true`	`false`
`10 == 10/3 * 3 + 10%3`	`true`
`10.2 / 2 == 5.1`	`true`
`(i > 0)? 1 : -1`	1 if `i>0`, -1 if not
`2 + " donuts"`	`"2 donuts"`
`i=3 \\ side effect i==3`	`3`
`i *= 2`	has value $2 * i$
`++i`	increases i, has value $i + 1$
`i++`	increases i, has value i

$\star$ whenever possible, use code which is as clear as possible

Priority of operations:

Documentation: `https://introcs.cs.princeton.edu/java/11precedence/`

Instructions

⋆ Bloc of instruction: between $\{, \}$ executes instructions in order

⋆ Expression based: evaluate the expression (with eventual side effect)

```
x=2;
i++;
<obj>.<meth>(<args>);
new SomeClass(<args>);
```

⋆ Delcarations

```
int x;
int y = 2;
final double pi = 3.14159;
```

Instructions: if-else

- ⋆ If-else instruction: `if (<expr bool>)I1 else I2`
- ⋆ If boolean expresion is `true` then execute the insctructions in `I1`
- ⋆ Otherwise execute instructions in `I2`
- ⋆ What does this instruction do?

```
if (x >= 0) {
    y = x;
} else {
    y = -x;
}
```

Instructions: switch

⋆ a "more complex if-else"
Example: (test for `i==0`, `i==1`, `i==2`)

```
switch (2*i) {
    case 4:
        System.out.print("much");
    case 1+1:
        System.out.println("too much");
}
```

⋆ evaluates all expressions (after switch and case). The expressions after case should be pairwise distinct
⋆ if there exists an expression after `case` which equals the expression after `switch` then all instructions between `:` and the last `}` of the switch are executed
⋆ if the `default` case is present then all instructions after default are executed.
⋆ if default is not present and no expression equals the tested expression then no instruction is executed.

Instructions: while

⋆ The while instruction: `while (<expr bool>)I`
Evaluates `<expr bool>`

- if `true` then execute the instructions in `I` and restart
- if `false` then stop

⋆ if `x` is an array with at least `n` elements then interpret the following:

```
int i = 0;
while (i < n) {
    System.out.print(((i == 0) ? "" : " ") + t[i]);
    i += 1;
}
```

Instructions: do-while

- ⋆ The do-while instruction: `do I while (<expr bool>);`
- ⋆ Execute the instruction `I` then evaluate the expression `<expr bool>`
  - if `true` then restart
  - if `false` then stop

```
int i = 0;
do {
    System.out.print(((i == 0) ? "" : " ") + t[i]);
    i += 1;
} while (i < n);
```

For loops

⋆ The for loop instruction: `for(<expr init>; <expr bool>; <expr maj>)I`

- Evaluate `<expr init>`
- (a) Evaluate `<expr bool>`
  - if `true`: Execute `I` then `<expr maj>` and restart at (a)
  - if `false`: then stop

```
for (int i = 0; i < n; i += 1) {
    System.out.print(((i == 0) ? "" : " ") + t[i]);
}
```

The break instruction

⋆ Break instruction: `break;`
⋆ finishes immediately the instruction inside which it is found (`switch`, `while`, `do`, `for`)

The continue instruction

⋆ `continue;`

Finishes immediately the current iteration and starts the next one

```
for (int i = 0; ; i++) {
    if (i == 5) {
        System.out.print("jump and "); continue;
    } else if (i == 10) {
        System.out.print("we're done !"); break;
    }
    System.out.print(i + " and ");
}
System.out.println();
```

The return instruction

⋆ `return [<expr];`

Finishes immediately the execution of a procedure (without `<expr>` or a function (with `<expr>`)

```
void showDiv(int x, int y) {
if (y == 0) {
    return;
}
    System.out.println(x / y);
}
```

```
String miroir(String s) {
    if (s.equals("")) {
        return "";
    }
    return
        miroir(s.substring(1))+ s.charAt(0);
}
```

Declaring a local variable

```
int x = 5;
int a, b = 3, c = b*2;
int[] d, e = new int[3];
int[][] t = new int[][] {{1,2,3},{4,5,6}};
final float pi = 3.14f;
```

⋆ Local variable: defined inside a bloc using a declaration with or without initialization
⋆ the visibility of the local variabile is limited to the block of definition

```
int n = 0;
while (n < 10) {
    int x = 5;          // x is defined in the while loop
    ...
    n += 1;
}
System.out.println(x); // x not accessible here!!
```

Rules for declaring local variables

- Every local variable must be **declared and initialized before it is used**
- A declaration can intervene **at every moment in a block**
- A variable is visible **inside the block** where it is declared, **after it is declared**
- A declaration **cannot contain another one**
- A constant (declared with `final`) can be initialized **only once**. Its value cannot change.