Course of Programming in Java

by Łukasz Stafiniak

at Institute of Computer Science
University of Wrocław

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

The Java Programming Language

Elements of chapter 7 to chapter 10

by Ken Arnold, James Gosling, David Holmes

Recommended Readings:

- *Java (programming language)* Wikipedia entry
  `http://en.wikipedia.org/wiki/Java_(programming_language)`

- Novice programmer textbook: *Introduction to Programming in Java: An Interdisciplinary Approach* by Robert Sedgewick, Kevin Wayne

- ***The Java Programming Language*** by Ken Arnold, James Gosling, David Holmes

- ***The Java Language Specification*** at Sun/Oracle.

  `http://java.sun.com/docs/books/jls/`

- ***The Java Tutorials*** at Sun/Oracle

- *Learning Java* at Wikiversity, or

  Java Programming at Wikibooks.

- Java SE6 API and Java SE7 API.

  `http://download.oracle.com/javase/7/docs/api/`

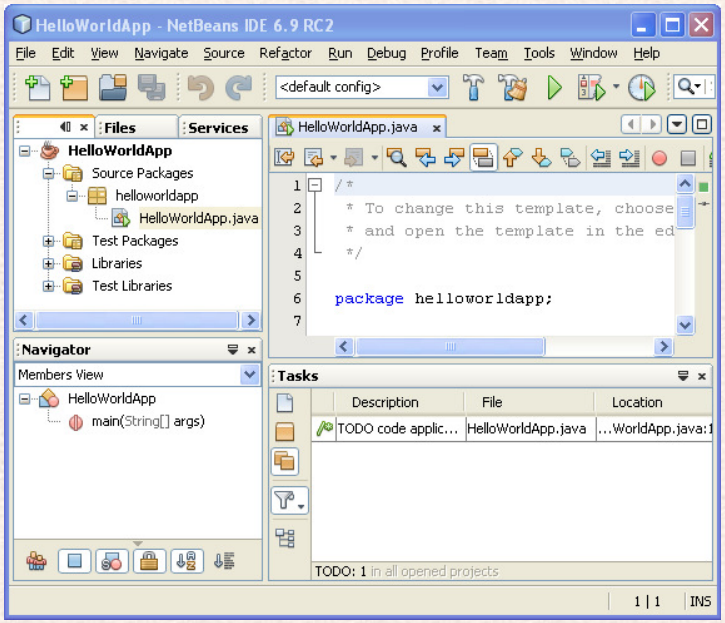
- ***Thinking in Java*** by Bruce Eckel.

- ***Effective Java*** by Joshua Bloch.

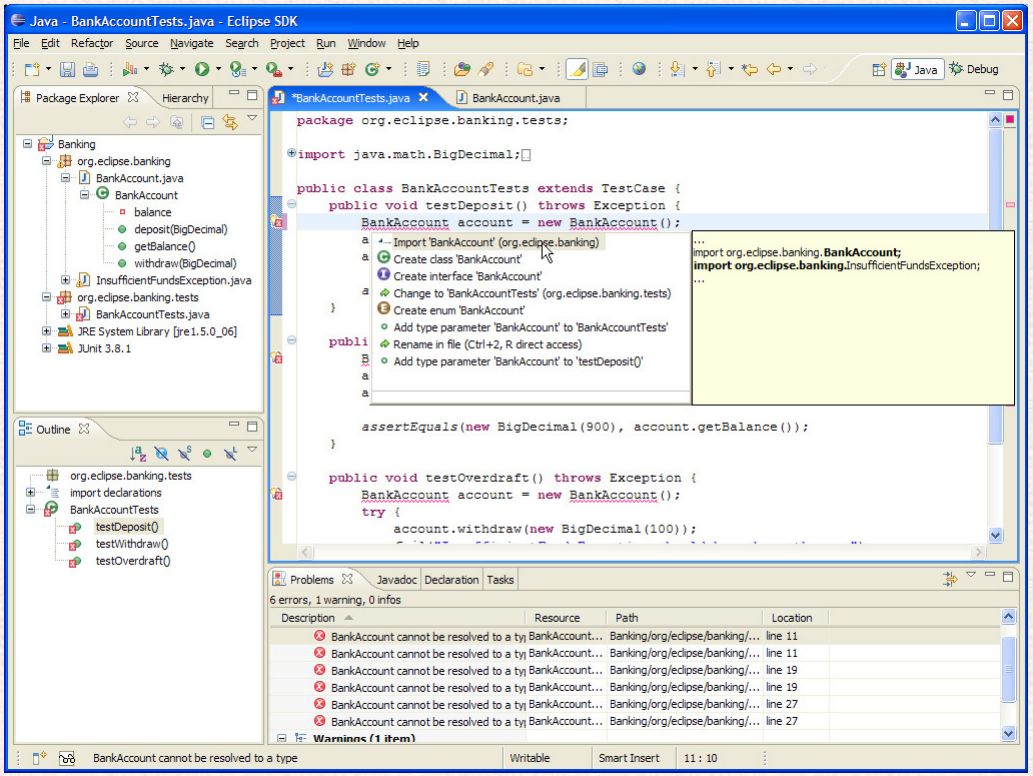
- ***Head First design patterns*** by Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates.

Integrated Development Envs

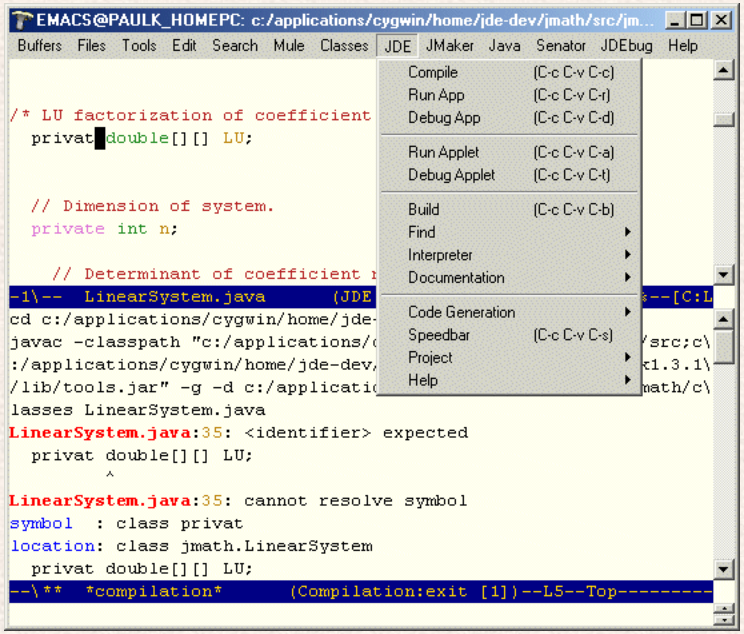
- **NetBeans** (for Java)

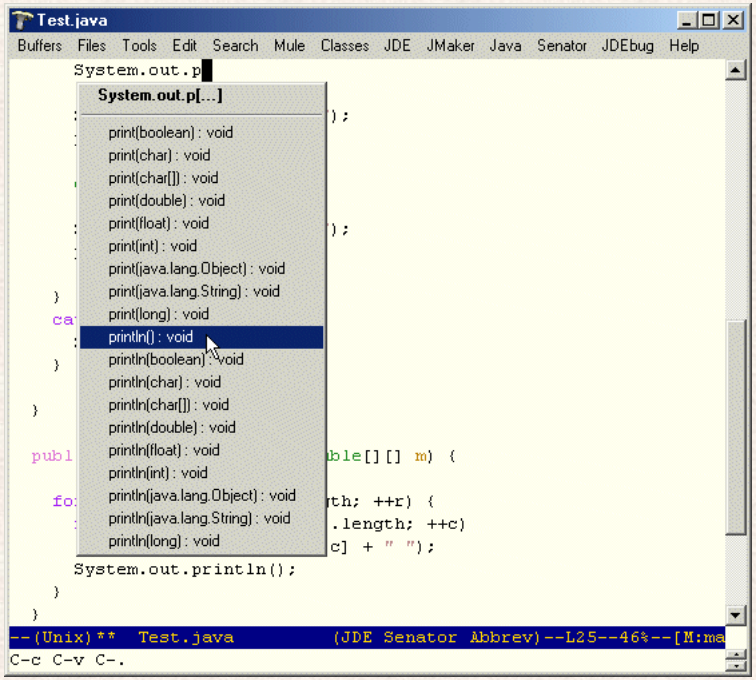


- **Eclipse** (general)



- **Emacs** with *CEDET* (general) and *JDEE*





Learning **NetBeans** is part of the course, but otherwise you should use (and master) your editor of choice.

Java History

- **1990**: Sun Microsystems decide to program consumer-electronic applications; James Gosling designs language called Oak, interpreted on a VM for portability across hardware

- **1991**: first prototype (mostly for interactive television)

- **1995**: World-Wide-Web blossoms; Oak renamed to Java and switching niche to running code (*applets*) in browsers

- earlier in Sun labs: Self (similar to Smalltalk) uses Just-In-Time compilation (JIT) achieving "50% of C efficiency"

  1999: HotSpot a JVM with JIT

- **2004**: many language features in Java 5, most important: generics (allow static typing of collections)

Features, Differences with C++

- Java is almost fully Object-Oriented while C++ is procedural+OO
- no hidden type conversions in Java
- no memory leaks and no segfaults due to dangling pointers, thanks to Garbage-Collector

  no destructors (actually, renamed to *finalizers* and rarely used) because no guarantee that objects deleted

- designed for concurrency from the start
- rich standard library and rich GUIs

Syntax: values and expressions

Keywords

The following table lists the keywords (keywords marked with a * are reserved but currently unused):

`abstract`	`continue`	`for`	`new`	`switch`
`assert`	`default`	`goto`*	`package`	`synchronized`
`boolean`	`do`	`if`	`private`	`this`
`break`	`double`	`implements`	`protected`	`throw`
`byte`	`else`	`import`	`public`	`throws`
`case`	`enum`	`instanceof`	`return`	`transient`
`catch`	`extends`	`int`	`short`	`try`
`char`	`final`	`interface`	`static`	`void`
`class`	`finally`	`long`	`strictfp`	`volatile`
`const`*	`float`	`native`	`super`	`while`

`null`, `true`, and `false` are formally literals, just like numbers.

Types and literals

The primitive data types are:

`boolean`	either `true` or `false`
`char`	16-bit Unicode `UTF-16` code unit (unsigned)
`byte`	8-bit signed two's-complement integer
`short`	16-bit signed two's-complement integer
`int`	32-bit signed two's-complement integer
`long`	64-bit signed two's-complement integer
`float`	32-bit `IEEE` 754 floating-point number
`double`	64-bit `IEEE` 754 floating-point number

and corresponding wrapper classes `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`.

Integer literals are of type `int` unless:

- they end in `L` or `l`, then `long`;
- small integer literals can be assigned to `short` / `byte` variables.

Floating-point literals are of type `double` unless end in `f` or `F`: `float`.

Wrapper classes and corresponding numeric values are convertible:

```
Integer val = 3; // boxing
int x = val; // unboxing
```

Given the method

```
static boolean sameArgs(Integer a, Integer b) {
    return a == b;
}
```

the following invocation returns `true`:

```
sameArgs(3, 3)
```

while this invocation returns `false`:

```
sameArgs(3, new Integer(3))
```

String literals are references an object of type `String`.

- Equal string literals produce the same `String` object.
- Method `intern()` produces at run-time the same object as produced for string literals.
- Strings are not mutable (they are not treated as arrays of characters).

Variables

```
int x;      // uninitialized, can't use
int y = 2;
x = y * y; // now x has a value
int z = x; // okay, safe to use x
```

Local and parameter variables cease to exist when the flow of control reaches the end of the block in which they were declared, any referenced object is under garbage collection rules.

Final local and field variables

```
final int id = nextID++;
```

Blank final variables are sometimes useful (compiler ensures correctness):

```
class NamedObj {
    final String name;
    NamedObj(String name) {
        this.name = name;
    }
}
```

```
final int[] numbers = numberList();
final int maxNumber; // max value in numbers
    int max = numbers[0];
    for (int num : numbers) {
        if (num > max)
            max = num;
    }
    maxNumber = max;
```

Hiding is not permitted in nested scopes within a code block:

```
{
    int über = 0;
    {
        int über = 2; // INVALID: already defined
        // ...
    }
}
```

but non-nested blocks or non-nested `for`-loops can use repeating names.

Use of **unicode** in identifiers is OK.

Arrays

Arrays of arrays: `float[][] mat = new float[4][4];`

same as

```
float[][] mat = new float[4][];
for (int y = 0; y < mat.length; y++)
    mat[y] = new float[4]; // initialized to 0.0
```

Array initialization

```
Attr[] attrs = new Attr[12]; // initialized to null

for (int i = 0; i < attrs.length; i++)
    attrs[i] = new Attr(names[i], values[i]);

int[][] pascalsTriangle = {

            { 1 },
            { 1, 1 },
            { 1, 2, 1 },
            { 1, 3, 3, 1 },
            { 1, 4, 6, 4, 1 },
        };
```

Anonymous array:

```
printStrings(new String[] { "one", "two", "many" });
```

Arrays and subtyping

Let `Y` be a subclass of `X`.

```
Y[] yArray = new Y[3];        // a Y array

X[] xArray = yArray;          // valid: Y is assignable to X
xArray[0] = new Y();
xArray[2] = new X();          // INVALID: can't store X in Y[]
xArray[1] = new Z();          // INVALID: can't store Z in Y[]
```

Arrays behave like `final` classes:

```
class ScaleVector extends double[] { // INVALID

    // ...
}
```

Operators

Arithmetic

+	addition
-	subtraction
*	multiplication
/	division
%	remainder

Increment and Decrement

```
class IncOrder {
    public static void main(String[] args) {
        int i = 16;
        System.out.println(++i + " " + i++ + " " + i);
    }
}
```

The output is `17 17 18`.

Relational and Equality Operators

`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to
`==`	equal to
`!=`	not equal to

Logical operators

`&`	logical `AND`
`|`	logical inclusive `OR`
`^`	logical exclusive or (`XOR`)
`!`	logical negation
`&&`	conditional `AND`
`||`	conditional `OR`

Bit manipulation operators

`&` bitwise `AND`
`|` bitwise inclusive `OR`
`^` bitwise exclusive or (`XOR`)

`<<` Shift bits left, filling with zero bits on the right-hand side
`>>` Shift bits right, filling with the highest (sign) bit on the left-hand side
`>>>` Shift bits right, filling with zero bits on the left-hand side

Conditional operator

```
value = (userSetIt ? usersValue : defaultValue);
```

is equivalent to

```
if (userSetIt)
    value = usersValue;
else
    value = defaultValue;
```

Assignment operators

Right-associativity lets assignments be chained together to give a set of variables the same value:

```
x = y = z = 3;
```

but the result of an assignment, e.g. `(z = 3)`, is a value, not a variable.

Given the variable `var` of type `T`, the value `expr`, and the binary operator op, the expression (where op= is +=, -=, *=, /=)

```
var op= expr
```

is equivalent to

```
var = (T) ((var) op (expr))
```

Precedence

Operators with the same precedence appear on the same line of the table:

postfix operators	`[] . (params) expr++ expr-`
unary operators	`++expr -expr +expr -expr ~ !`
creation or cast	`new (type)expr`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > >= <= instanceof`
equality	`== !=`
AND	`&`
exclusive OR	`^`
inclusive OR	`|`
conditional AND	`&&`
conditional OR	`||`
conditional	`?:`
assignment	`= += -= *= /= %= >>= <<= >>>= &= ^= |=`

String concatenation

The + operator is interpreted as the string concatenation operator whenever at least one of its operands is a `String`. If only one of the operands is a `String` then the other is implicitly converted to a `String` via the `toString` method. `null` is converted to string as `null`.

Order of Evaluation

Left to right. With exception of arguments to `&&` and `||`, all subexpressions are evaluated unless some evaluation terminates abruptly (e.g. exception).

Syntax: statements

Not all expressions can be turned into statements by adding the terminator `;`

Only:

- Assignment expressions (= or one of the `op=` operators)
- Prefix or postfix forms of ++ and --
- Method calls (whether or not they return a value)
- Object creation expressions (using `new` to create an object)

```
ifelse

public void setProperty(String keyword, double value)
    throws UnknownProperty
{
    if (keyword.equals("charm"))
        charm(value);
    else if (keyword.equals("strange"))
        strange(value);
    else
        throw new UnknownProperty(keyword);
}
```

switch

Here using enum constants.

```
Verbose v = ... ; // initialized as appropriate
public void dumpState() {
    int verbosity = v.getVerbosity();
    switch (verbosity) {
      case Verbose.SILENT:
        break; // do nothing
      case Verbose.VERBOSE:
        System.out.println(stateDetails);
        // FALLTHROUGH
      case Verbose.NORMAL:
        System.out.println(basicState);
        // FALLTHROUGH
      case Verbose.TERSE:
        System.out.println(summaryState);
        break;
      default:
        throw new IllegalStateException(
                      "verbosity=" + verbosity);
    }
}
```

while and dowhile, C-like for

```
while (hi < MAX) {
    System.out.println(hi);
    hi = lo + hi;
    lo = hi - lo;
}
```

```
do
statement
while (expression);
```

```
for (int i = 0, j = arr.length - 1; j >= 0; i++, j--) {
    // ...
}
```

Enhanced `for` statement

```
for (Type loop-variable : set-expression)
statement
```

`set-expression` must either be an array, or an object that implements the interface `java.lang.Iterable`.

```
static double average(int[] values) {
    if (values == null || values.length == 0)
        throw new IllegalArgumentException();

    double sum = 0.0;
    for (int val : values)
        sum += val;

    return sum / values.length;
}
```

`break` and `continue`, also with labels

```
public boolean workOnFlag(float flag) {
    int y, x;
  search:
    {
        for (y = 0; y < matrix.length; y++) {
            for (x = 0; x < matrix[y].length; x++) {
                if (matrix[y][x] == flag)
                    break search;
            }
        }
        // if we get here we didn't find it
        return false;
    }
    // do some stuff with flagged value at matrix[y][x]
    return true;
}

while (!stream.eof()) {
    token = stream.next();
    if (token.equals("skip"))
        continue;
    // ... process token ...
}
```

```
return

protected double nonNegative(double val) {
    if (val < 0)
        return 0;   // int constant can be assigned to double
    else
        return val; // a double
}
```