IST-4-JAV Java Programming
Class 1 — (re ?)Discovering Java

Alice BRENON <alice.brenon@liris.cnrs.fr>

LIRIS
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Foreword: the IST-4-JAV course

Timetable

20h: 5×4h-sessions

- 2023-10-02 8 a.m. (today!)
- 2023-10-04 2 p.m.
- 2023-10-06 8 a.m.
- 2023-10-11 2 p.m.
- 2023-10-13 8 a.m.
- (one class every other day this week, same last week except monday)

Time repartition

- ~2h course
- (a break in-between)
- ~2h practice

Course home

https://perso.liris.cnrs.fr/abrenon/IST-4-JAV.html

How to pass this class ?

Evaluation

Game project demonstrating the object-oriented concepts seen in class

Time-budget

- 20h together in class
- 20h at home
  - 1h½ after each class
  - 12h½ project

About programming

Modeling things

Back to maths 101

What is a *number* ?

Back to maths 101

What is a *number* ?

Let's consider: **3**

- $\triangle$ (geometric property) ?

Back to maths 101

What is a *number* ?

Let's consider: **3**

- $\triangle$ (geometric property) ?
- $\{a, b, c\}$ (set) ?

Back to maths 101

What is a *number* ?

Let's consider: **3**

- △ (geometric property) ?
- $\{a, b, c\}$ (set) ?
- $succ(succ(succ(0)))$
  (Peano arithmetic) ?

Back to maths 101

What is a *number* ?

Let's consider: **3**

- $\triangle$ (geometric property) ?
- $\{a, b, c\}$ (set) ?
- $succ(succ(succ(0)))$ (Peano arithmetic) ?
- $\frac{51}{17}$ (result of a computation) ?

Back to maths 101

What is a *number* ?

Let's consider: **3**

- $\triangle$ (geometric property) ?
- $\{a, b, c\}$ (set) ?
- $succ(succ(succ(0)))$ (Peano arithmetic) ?
- $\frac{51}{17}$ (result of a computation) ?

Base b

- `11` (base 2)

Back to maths 101

What is a *number* ?

Let's consider: **3**

- $\triangle$ (geometric property) ?
- $\{a, b, c\}$ (set) ?
- $succ(succ(succ(0)))$ (Peano arithmetic) ?
- $\frac{51}{17}$ (result of a computation) ?

Base b

- `11` (base 2)
- `3` (base 13)

Back to maths 101

What is a *number* ?

Let's consider: **3**

- $\triangle$ (geometric property) ?
- $\{a, b, c\}$ (set) ?
- $succ(succ(succ(0)))$ (Peano arithmetic) ?
- $\frac{51}{17}$ (result of a computation) ?

Base b

- `11` (base 2)
- `3` (base 13)

$$\sum_{i=0}^{\infty} c_i * b^i$$

Non-positional systems

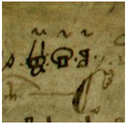


Figure 1: Glagolitic numerals (1280)

Figure 2: Roman numerals (1909)

$$I + II = III \qquad \text{easy!}$$
$$I + IV = V \qquad \text{uh ?}$$
$$XCV + V = C \qquad \text{haha good one romans ^^}$$

- hard to write addition rules
- not digits, numbers ('X' ≠ 'C' vs. '1' in 10 = '1'in 100)
- limited (no symbol > 1000)

The advantage of positional systems

- finite set of simple ("mechanical") rules
- can represent any number, even one you've never even considered

$132 + 41 = ??$

The advantage of positional systems

- finite set of simple ("mechanical") rules
- can represent any number, even one you've never even considered

$132 + 41 = ??$

…in base 5!

Back to elementary school!

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	+1 0
2	2	3	4	+1 0	+1 1
3	3	4	+1 0	+1 1	+1 2
4	4	+1 0	+1 1	+1 2	+1 3

$$\begin{array}{r} 132 \\ +\ \ 41 \\ \hline \end{array}$$

Back to elementary school!

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	+1 0
2	2	3	4	+1 0	+1 1
3	3	4	+1 0	+1 1	+1 2
4	4	+1 0	+1 1	+1 2	+1 3

$$\begin{array}{r} 132 \\ +\ \ 41 \\ \hline \end{array} \qquad \begin{array}{r} 132 \\ +\ \ 41 \\ \hline 3 \end{array}$$

Back to elementary school!

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	+1 0
2	2	3	4	+1 0	+1 1
3	3	4	+1 0	+1 1	+1 2
4	4	+1 0	+1 1	+1 2	+1 3

$$\begin{array}{cccc} & 1 & 3 & 2 \\ + & & 4 & 1 \\ \hline & & & \end{array} \qquad \begin{array}{cccc} & 1 & 3 & 2 \\ + & & 4 & 1 \\ \hline & & & 3 \end{array} \qquad \begin{array}{cccc} & 1 & & \\ & 1 & 3 & 2 \\ + & & 4 & 1 \\ \hline & & 2 & 3 \end{array}$$

Back to elementary school!

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	+1 0
2	2	3	4	+1 0	+1 1
3	3	4	+1 0	+1 1	+1 2
4	4	+1 0	+1 1	+1 2	+1 3

$$\begin{array}{r} 132 \\ +\quad 41 \\ \hline \end{array} \qquad \begin{array}{r} 132 \\ +\quad 41 \\ \hline 3 \end{array} \qquad \begin{array}{r} 1 \\ 132 \\ +\quad 41 \\ \hline 23 \end{array} \qquad \begin{array}{r} 1 \\ 132 \\ +\quad 41 \\ \hline 223 \end{array}$$

Let's check!

$$132 = 2 * 5^0 + 3 * 5^1 + 1 * 5^2 = 2 + 15 + 25 = 42$$

$$41 = 1 * 5^0 + 4 * 5^1 = 1 + 20 = 21$$

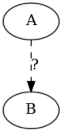
$$223 = 3 * 5^0 + 2 * 5^1 + 2 * 5^2 = 3 + 10 + 50 = 63$$

\\^θ^/

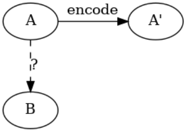
A *model* of numbers

- a concept: *numbers*
- a representation: *digits*
- arithmetic rules to handle digits
- → know how to write a number with digits: *encode*
- ← know what number digits represent: *decode*

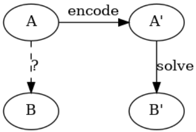
General pattern



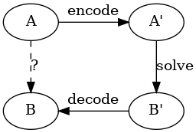
General pattern



General pattern



General pattern



The core of programming

- defining abstract concepts from a concrete implementation (the **right** level)
  - too complex: it's slow
  - too simple: it's hard to use
- solve problems using the abstraction
- have a system translate it to the implementation
- repeat
- run

Layers

- higher-level languages
- C
- assembly
- machine binary

↕ compilation / interpretation

Expressing computations

Imperative

- “do things in a given order”
- recipe
- implicit reference to a state

Imperative

- "do things in a given order"
- recipe
- implicit reference to a state

```
for(int i = 0; i < 4; i++)
    a[i] += 1;
}
```

Functional

- "describe the computation itself"
- based on lambda-calculus
- everything is a function ($\Rightarrow$ higher-order)

Functional

- "describe the computation itself"
- based on lambda-calculus
- everything is a function (⇒ higher-order)

```
fmap (+1) positions
```

Object

- “as a metaphor of a physical object”
- associate data and logic
- explicit reference to an identified state

```
for(Cell cell : cells) {
    cell.incr();
}
```

Anything else ?

Anything else ?

- logic:

```
sum(s(a), b) :- sum(a, s(b))
```

Anything else ?

- logic:

```
sum(s(a), b) :- sum(a, s(b))
```

- concatenative:

```
: fac 1 swap 1+ 1 ?do i * loop ;
```

Anything else ?

- logic:

```
sum(s(a), b) :- sum(a, s(b))
```

- concatenative:

```
: fac 1 swap 1+ 1 ?do i * loop ;
```

- (machine learning ?)

Compiling vs. Interpreting

Compiler

- generate low-level from high-level
- optimized, fast

Interpreter

- translated on the fly (duality program / data)
- generally slower (+ loading time)
- portable!

Typing

Labels on things in the memory:

- “strong” or “weak”
- explicit or implicit
- static or runtime
- more or less expressive
  - `void*`
  - (G)ADT
  - entire logic system

Actually running them

A finite memory

- hopefully “big enough”
- representing *numbers*

A finite memory

- hopefully “big enough”
- representing *numbers*

Figure 3: An abacus

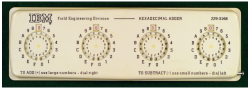


Figure 4: A modern Pascal’s calculator

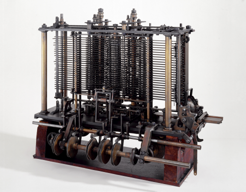


Figure 5: The Analytical Engine

A finite memory

- hopefully “big enough”
- representing *numbers*

Figure 3: An abacus

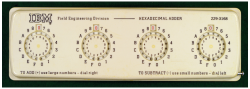


Figure 4: A modern Pascal’s calculator

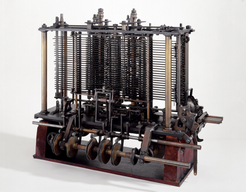


Figure 5: The Analytical Engine

- which can represent *things*

Anything is a number

Directly ("native")

- **Truth value** True or False, 2 values → 1, 0
- **Numbers** obvious **but** overflow
- **Characters**
  - very natural (remember non-positional systems ?)
  - known for very long (before Cæsar)!
  - → encodings (ASCII, UTF-8. . .)

As a sequence

- notion of address
- special strategies: "stop" symbol vs. length
- "large" numbers
- text
- multimedia

The right word

An *atomic* number ≐ a *word*

How to choose the right bits size

- too large is painful to build
- too small is painful to use (slow)

Instructions

- the **paths** in the circuit
- at the **electrical** level
- **hardwired** operations
  - sum
  - multiplication
  - bit shift
  - xor
  - ...

Architecture

word size + set of instructions = a “machine”

Examples

- x86_64
- arm64
- i686
- riscv64

Virtual architecture

- data: *numbers*
- operations: *numbers*

⇒ we can have *programs* pretending to be *machines* (see Turing machines)

Java

Concepts

Compiled or interpreted ?

- compiles to a binary: *bytecode*...
- but for a *virtual machine*! “JVM”
- “Write Once Run Anywhere”

Features

- Object-Oriented (+ Imperative)
- strictly typed: forget that “anything is a number”
- rich collection of built-ins for data structures, I/O...
- automatic memory handling (garbage collector)

History

Context

- released in 1995 (32-bits architectures)
- after the Eternal September!

Internet oriented

- support from Netscape
- the .com frenzy
- applets, "servlets" (e-commerce, administrations. . .)

(Big) Business

- created by Sun Microsystem, bought by Oracle
- IDE (NetBeans, Eclipse), "easy", "predictible" (developer as a "worker")
- a Java "Enterprise Edition" (vs. JSE)
- a lot of marketing, "Java" meant "cool" (→ "javascript")

Language basics

Language basics

Contains real bits of Java.

- `this` is for actual valid code
- `<THIS>` is for meta bits of code (templating)
- mind the case, the quotes, etc.

Types

What they are

How to navigate the “everything is a number” soup ?

What they are

How to navigate the “everything is a number” soup ?

→ *flags*

- boundaries (size)
- purpose, intention
- ~ sets in maths
- prevent (some) errors

Usage

Convention

- native (lowercase)
- sequences (uppercase-first)
- `void`

Every value or function

must be annotated with its type (Java does it too and compares)

A REPL!

In `jshell`[1] (Read - Eval - Print Loop) type

- */vars* : to print the known variables with their type
- */set feedback verbose* : to include the type of expressions in the evaluation output

This is **not Java**! only special commands for the interpreter

[1] https://tryjshell.org/

Data structures

“native” numbers

Truth values

- `boolean` useful for conditionals

Examples

- `true`
- `false`

Integers

- `byte` 8 bits integers → $[-128, 127]$
- `int` 32 bits integers → $[-2147483648, 2147483647]$
- `short` 16 bits integers → $[-32768, 32767]$
- `long` 64 bits integers → $[-2^{63}, 2^{63}]$

Examples

- `0`
- `-1`
- `1347` (not for byte)
- `0b11` (binary), `046` (octal), `0xa3` (hexadecimal)

Decimal numbers

- `float` 32 bits decimal numbers, scientific notation, significand/exponent
- `double` same with 64 bits (+ precision)

Examples

- the previous (since $\mathbb{N} \subset \mathbb{R}$)
- `1.03, -0.47, 320.`
- `314e-2, -1.21E7`
- `1f, 2.0d, -1.237e12F`

Unicode characters

- `char` 16 bits, UTF-16, written between simple quotes

Examples

- `'a','b','0','!','@','é'...`
- `'\n','\r','\t'...`
- `'\'','\\'`
- `'\u00e9'` (code point)
- `10,0x27`

“sequences” = objects

“Arbitrary” precision

- `BigInteger` large integers
- `BigDecimal` large decimal numbers

Text

- `String` immutable sequences of characters

Examples

- `""`
- `"Some text"`
- `"first line\nsecond line\n"`

Values

Constants

- (everything we've just seen)
- "magic" values, not all explicitly defined (because: `long`, `String`)
- a very special constant only for objects: `null`

Problems with constants

- "Don't Repeat Yourself"
- they don't carry any *intention* ("beware of names" said I!!)
- "constants" change sometimes (e.g. exchange rate)

Variables

Concept

- give a **meaningful** name to a value
- absolutely abstract, will need to refer to a place in the memory
- a “wire”, “bringing” the value (*no copy*)

Valid names

- must start by: a letter (`[a-zA-Z]`), `$` or _
- may contain any above and digits `[0-9]`
- except reserved keywords: `if`, `else`, `for`, `while`, `return`, `try`, `catch`, `static`, `final`...
- usually: full uppercase for “constant” variables, camelCase for the rest

Built-in (`final`) variables

- streams: `System.in`, `System.out`
- maths: `Math.PI`, `Math.E`

Comments

Comments

- clarify intent
- not a remedy for bad naming
- not needed to caption the obvious
- can contain the documentation (*javadoc*)

Syntax

Rest of line

```
// this is a comment
4 // it doesn't have to start with the line
// but it ends it so this is not a number: 17
```

General comments

```
/* these comment can start on a line
   and end on another one */
/* Since they have an end, this is a number: */ 4
/** two asterisks like this for a Javadoc string */
```

Functions and Procedures (built-ins only today)

Functions

- abstracts a computation by isolating its inputs
- name it ("beware of names"!)
- has several input types (for its arguments) and one output type (for its result)
- its arguments are written within parentheses (both in declaration and call)

Built-in functions examples

- **type conversions** `Integer.parseInt`, `Integer.toString`...
- **maths toolbox** `Math.max`, `Math.min`, `Math.exp`...
- ...

Procedures

- simply "do" something
- no result
- (actually has special output "type" `void`)

Built-in procedures

- `Thread.sleep` (pause execution)
- `System.exit` (quit the program)

Operators (all built-ins)

- special functions with an infix notation
- name: a few punctuation/typographic characters

Unary

- `~`, `!`

Binary

- numbers: `*`, `-`, `/`, `%`
- numbers and `String`s: `+`
- boolean: `||`, `&&`
- bitwise: `|`, `&`, `>>`, `<<`, `^`
- comparisons: `==`, `!=`, `>`, `<`, `>=`, `<=`

Methods

- functions or procedures
- tied to an object by a `.`
- names need not be unique

Examples

- `System.out.println`
- `"some string".length`
- `userName.charAt`
- `password.equals`

About the memory

Value is just a (JVM, not physical) `word`

- "Numbers" → their direct value
- objects → address in memory
- (can be nested, so it's just a graph of pointers)
- no direct access but: binary operators, variables

Consequences

- only objects can be `null`
- but **`null` isn't an object**
- `==` / `!=` on objects compare addresses

Expressions and Statements

Simple bricks ("atoms")

Expressions

- compute a value
- constants
- variables (in a context where they are defined — "plugged wire")
- any other type

Statements

- do something (change state)
- only "atomic" statement: variable declaration
- type `void`

Declaring a variable

- reserve space in memory
- tag it with a given type
- can be set with an initial value, but always initialized
  - “numbers”: to the equivalent of `0`
  - objects: to `null`

Examples

```
int messageLength;
String userName;
char firstLetter = 'a';
```

Nesting

Expressions

- if `e` is an expression, so is `(e)`
- (useful for operators priority)

Statements

- if `s1` and `s2` are statements, `s1; s2` is a statement
- semantics: `s1` then `s2`
- in practice, wrap all the list within `{...}`

```
{ s1; s2; s3; ... ; sn }
```

Casts

Use

- force a conversion between types
- may lose information (beware of truncation)
- (give hints to Java)

assuming

- `t` is a type
- `<VALUE>` is an expression

```
(t) <VALUE>
```

is an **expression** of type `t` with value "projected" from `<VALUE>`

Casts examples

```
(short) 4 // still 4, but coded on 16 bits
(byte) 'c' // == 99
(float) (2.5 / 2) // still 1.25, but as a float
(int) 7.2 // truncates to 7
(int) 7.9 // 7, it truncates and doesn't round
(float) ((int) 7.2) // still 7.0, information was
                    // lost
```

Function application

- a call to a function or an operator is an **expression**
- a call to a procedure is a **statement**

Syntax

- function or procedure: its name followed by the comma-separated arguments between parentheses
- operators: the symbol before, between or after its argument(s)

Examples

```
!false
total / count
"Hello, " + "world!"
Math.pow(Math.E, -1)
System.out.println(someMessage);
```

Variable assignment

an expression which

- changes a value in the memory
- returns a value

assuming

- `a` is a variable of type `t`
- `<VALUE>` is an expression of type `t`

```
a = <VALUE>
```

is an **expression** wich assigns the value of `<VALUE>` to `a` and has the same value

Assignment operators

- **Binary** the previous binary (except `boolean`) operators followed by `=` (`+=`, `*=`, `/=`, `|=`, `<<=`...)
- **Unary**
  - shortcuts for the pattern `a = a · 1` (·: '+' or '-')
  - *increment*: `++` / *decrement*: `--`
  - before or after the variable: value after or before the change

```
int a = 4;
++a; // a is now 5, the line evaluates to 5
--a; // a is back to 4, the line evaluates to 4
int b = a++; // read then increment: a is now 5
             // b is 4
a--; // a is back to 4 again, the line evaluates
     // to 5
```

Control structures 101

Conditional statement

assuming

- `<TEST>` is an expression of type `boolean`
- `<WHEN_TRUE>` and `<WHEN_FALSE>` are statements

```
if(<TEST>) {
    <WHEN_TRUE>
} else {
    <WHEN_FALSE>
}
```

```
if(<TEST>) {
    <WHEN_TRUE>
}
```

are **statements** which:

- if `<TEST>` evaluates to `true`, execute `<WHEN_TRUE>`
- otherwise execute `<WHEN_FALSE>` (or nothing for the shorter form)

Conditional expression

assuming

- `<TEST>` is an expression of type `boolean`
- `<WHEN_TRUE>` and `<WHEN_FALSE>` are expressions

```
<TEST> ? <WHEN_TRUE> : <WHEN_FALSE>
```

is an **expression** which value is:

- `<WHEN_TRUE>` if `<TEST>` evaluates to `true`
- `<WHEN_FALSE>` otherwise

`for` loops

assuming

- `<INITIALIZE>`, `<ITERATE>` and `<BLOCK>` are statements
- `<TEST>` is an expression of type `boolean`

```
for(<INITIALIZE>; <TEST>; <ITERATE>) {
    <BLOCK>
}
```

is a **statement** which:

- executes `<INITIALIZE>`
- if `<TEST>` evaluates to `true`, executes `<BLOCK>` then `<ITERATE>` then start over from this line
- otherwise stops

`while` loops

assuming

- `<TEST>` is an expression of type `boolean`
- `<BLOCK>` is a statement

```
while(<TEST>) {
    <BLOCK>
}
```

is a **statement** which:

- if `<TEST>` evaluates to `true`, executes `<BLOCK>` and start over from this line
- otherwise stops

Using it

How things work in general

Reminder

- hybrid between compiled and interpreted
- compiles (“source”) code to binary
- binary is not for the physical architecture but for the Java Virtual Machine

Java projects

- organized in packages (folders)
- packages contain classes (files)
- classes contain “code”
  - values
  - methods

Names

The Operating System

- file system path
- separated by /
- `.java` extension

Java

- package / classes
- separated by .
- no extension

Compilation

Operating System → Java

- `static`
- `.java` extension
- compiled into bytecode `.class`
- may catch some mistakes

```
javac [OPTIONS] PATH_TO_SOURCE_FILE
```

Execution

- `dynamic, runtime`
- the `.class` run by the JVM
- may catch some other errors

```
java [OPTIONS] PATH_TO_BYTECODE
```

For today

Read the documentation

Main page of the official Java documentation

https://docs.oracle.com/en/java/javase/19/docs/api/index.html

Some particularly interesting modules and packages to start

- `java.base`
- `java.lang`

Example

String

All Methods	Static Methods	Instance Methods	Concrete Methods	Deprecated Methods

Modifier and Type	Method	Description
char	charAt(int index)	Returns the char value at the specified index.
IntStream	chars()	Returns a stream of int zero-extending the char values from this sequence.
int	codePointAt(int index)	Returns the character (Unicode code point) at the specified index.
int	codePointBefore(int index)	Returns the character (Unicode code point) before the specified index.

JShell documentation

https://cr.openjdk.org/~rfield/tutorial/JShellTutorial.html

Time for practice :)

https://perso.liris.cnrs.fr/abrenon/IST-4-JAV.html