IST-4-JAV Java Programming
Class 2 - Enough to fly

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

1. Making functions and procedures

2. More data!

3. More control structures!

4. Our first program

Making functions and procedures

Why?

Previously on IST-4-JAV…

```
int initValue = 221;
while(initValue > 1) {
    System.out.println(initValue);
    if(initValue % 2 == 0) {
        initValue /= 2;
    } else {
        initValue = 3*initValue + 1;
    }
}
```

- `initValue` is modified!
- → need to reset it
- naming issue: at the end it's actually the "final" value
- not *reusable* nor *composable*

Functions as an interface

Documentation

- identifies what is needed
- partly document things through types
- beware of the naming though!

Local variables

- “What happens in a function stays in the function“
- variables defined within are not available outside
- can even reuse name (dangerous though)
- notion of *scope*

How?

Example: is a `char` contained in a `String`?

We know how to

- compare 2 `char`s (==)
- access a `char` at a given index in a `String` (`.charAt`)
- iterate over a `String` (`for` loop)

Example: is a `char` contained in a `String`?

We know how to

- compare 2 `char`s (`==`)
- access a `char` at a given index in a `String` (`.charAt`)
- iterate over a `String` (`for` loop)

→ algorithm

```
for each index i within the input string s:
    compare the char c with the one at index i
    if they are equal
        return true
    otherwise
        keep going
endfor
return false
```

The `head` of a function

```
boolean contains(String haystack, char needle)
```

- `boolean`: output type
- `contains`: function name
- `(..., ...)`: arguments are defined as a tuple
- `String haystack`: each argument is defined by its type and its name → variable

A new statement: `return`

- only available in a function/method declaration
- stops the current function
- “eliminates” the value of an expression → a statement
- must match the type of the function (like declaring a variable)
- “closing the box”

Examples

Correct use

```
return 4; // ok in an int, double, etc., function
return; // ok in a procedure (void function)
```

Incorrect use

```
int n = return 2; // bad! won't compile
someFunction(return 2); // bad! won't compile
                        // either
```

The body

- “regular” (= *imperative+object*) code like in `jshell`
- “virtual” environment, “suspended” computation, “assumption”
- can declare variables, use statements, call other functions. . .
- every “exit” must be checked: `return;` by default, but that is `void`

Implementing the body

Context

- `needle`: the `char` we're looking for
- `haystack`: the `String` where we're searching

```
for(int i = 0; i < haystack.length(); i++) {
    if(haystack.charAt(i) == needle) {
        return true;
    }
}
return false;
```

Wrapping it up

```
boolean contains(String haystack, char needle) {
    for(int i = 0; i < haystack.length(); i++) {
        if(haystack.charAt(i) == needle) {
            return true;
        }
    }
    return false;
}
```

Meditations

Good practice

- give clear, meaningful names to your functions
- have them use rich types to clarify their purpose
- give clear, meaningful names to its arguments
- functions should do "one thing"

Purity vs. side effects

Maths, again!

$$f\colon \begin{array}{ccc} A & \rightarrow & B \\ x & \mapsto & f(x) \end{array}$$

- doesn’t “change” `x`
- stable: always returns the same thing

Side effects

Sometimes programs need to

- alter values
- interact with the outside world
- notion of “state” of the program
- `void` type

Comparison with other loops

Similarities

- “canned” code
- the body within `{ ... }`
- made of statements

Differences

- loops are directly applied
- you can chose function names
- they have a return type
- applying them makes expressions (function) or statements (procedure)

More data!

“Object-Oriented”

The world of Ideas

Figure 1: A “duck”

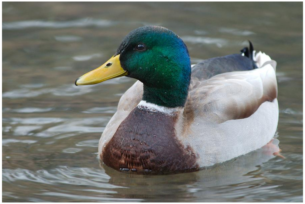

Figure 2: An other “duck”

Plato: the “Theory of Forms/Ideas”

- “true” reality of this world
- *eternal*, perfect
- they exist *prior to* particular concrete objects

Classes

- the “idea”, a “mold” to cast objects
- also, a “factory” to create particular realization

In Java

- *eternal*: compile-time (`static`), “structure” of your program
- *prior to* concrete objects: classes → objects
- not values **but** can appear in some expressions

The “Form“

- groups values together (≈ cartesian product, *AND*)

$$2DPoint = (\mathbb{N} \times \mathbb{N})$$

- defines a *recipe* for values: *what* and *how*
- defines the use of values
- similar to the *type* of “number” values (`int`, `char`...)
- you already know: `String` (mind the case!)

Fields

- define the *what*, the “components”
- needn’t be of the same type (≠ `String` ≈ `char` × `char` × . . . × `char`)
- just like in `jshell` : may set a value or not (always a default)

```
int room;
String building;
```

A “constructor”

- define the *how*, the “mold” part
- it’s just a function! (can take parameters)
- both the return type *and* the name of the function
- usually initializes the fields
- always a default constructor: no arguments, does nothing

```
ClassRoom(int room, String building) {
    this.room = room;
    this.building = building;
}
```

Methods

- like regular functions
- but linked to the class: can see
  - “inside” objects
  - other methods in the same class

```
String view() {
    return this.building.charAt(0)
        + "-" + this.room;
}
```

All together now

```
class ClassRoom {
    int room;
    String building;

    ClassRoom(int room, String building) {
        this.room = room;
        this.building = building;
    }

    String view() {
        return this.building.charAt(0)
            + "-" + this.room;
    }
}
```

Objects

- concrete realization of classes (one particular duck)
- "instance" of classes (*instantiate*)

In Java

- *runtime*: created on the fly while programs run
- they are values: can be stored, passed to / returned from functions
- instantiating an object takes memory

Creating objects

```
ClassRoom javaLab = new ClassRoom(17, "Hopper");
```

- `new` tells Java we're going to allocate some space
- the constructor tells Java which Class is needed
- an empty "box" is created
- the code for the constructor is called

(What’s `this`? What’s `this`? I can’t believe my eyes. . .)

```
Room(int room, String building) {
    this.room = room;
    this.building = building;
}
```

- the empty, fresh, “box” → `this`
- refers to the current instance
- ≈ 1st person pronoun

Using fields

```
javaLab.room // an int value (17)
javaLab.view() // a String value ("H-17")
```

assuming

- `objectExpression` is an expression of (object) type `T`
- `field` is exposed by class `T` as a property of type `output` or a method

```
objectExpression.field
```

is either

- an **expression** of type `output`
- or a **function** or **procedure**

`static`

- related to the "Form" itself, not to any of its instances
- *shared* space between instances
- ⟹ `static` code can refer to `this`!

Warning

fields and methods can be `static` ≠ constructors can (obviously) not

Syntax

`static` values can be initialized

- during declaration (preferred → no constructors!)
- within a method (`static` or not)

```
class ClassRoom {
    static String separator = "-";

...

    String view() {
        return this.building.charAt(0)
            + separator
            + this.room;
    }
}
```

Please note

- `static` values should only be accessed through the class itself, not one of its instances (though it works).

Recommended

```
ClassRoom.separator;
```

Not recommended

```
javaLab.separator;
```

Object tooling

Wrappers around native types

- `char` → `Character`
- `int` → `Integer`
- `double` → `Double`
- ...

useful for their `static` methods

- `Character.isLowerCase`
- `Integer.parseInt`

instanceof

assuming

- `a` is a variable of a class type
- `ClassName` is the name of an existing class

```
a instanceof ClassName
```

is an **expression** of type `boolean` which evaluates to `true` if `a` belongs to `ClassName`, `false` otherwise.

Example

```
String s = "";
Integer n = 4;
s instanceof String // == true
n instanceof Integer // == true
s instanceof Integer // will break at compile time!
```

Containers

Arrays

- several values of the same type put together
- indexed by an `int`
- (|| `String`)
- `.length`: special attribute but no methods

Creation

- type: content type suffixed with `[]`
  - `char[]`,`double[]`,`String[]`,`ClassRoom[]`
- value: either
  - empty (size only): `new char[5]`
  - pre-initialized: `{'a', 'b', 'c'}`

Warning

- pre-initialized form works only for declaration, not update
- otherwise, initialized by default → ≈ `0` (`null` for objects)

Access

once an array `a` of length `n` exists in memory it’s like:

- `n` *independent* variables exist
- each at index `i` can be:
  - read: `a[i]`
  - written: `a[i] = someNewValue;`
- its length is stored and readable: `a.length` (= `n`)
- (but not writable!)

Warning

- indices range from `0` to `n - 1`
- accessing an array out of its bounds will cause an error

Objects

- several pre-defined classes:
  - `ArrayList`
  - `LinkedList`
  - `Vector`
  - `...`
- different *strategies* to handle storage, grow, access, etc.

Creation

- type: content type between `<>`
  - `LinkedList<Integer>`
- value: with `new`, like any objects, empty or from another object

```
Vector<Integer> pages = new Vector<Integer>();
```

Access

no “cell as a variable” but

- `.get(int index)` returns the value at `index`
- `.set(int index, E element)` replaces the value at `index`
- `.add(E element)` appends (at the end)
- `.size()` (≠ `.length`!) the number of elements

"Boxes"

- not "one" type, an (open) "family"
- notion of type "variable" (`E`)
- any type → can be nested!

can be a "number" type or a class for arrays

```
int[] primes = new int[8];
String[] cheeseNames = {"camembert", "maroilles"};
int[][] matrix = new int[5][5];
```

must be a class for object containers

```
ArrayList<Integer> grades;
new LinkedList<String>()
```

Mutability

On values

- `String`: `.charAt()` function result → read-only
- arrays: `[]` variable → read-write
- object containers: `.get`, `.set` → read-write

On size

- `String`: can't be changed
- arrays: `.length` is `final` (ro)
- object containers: `.add`

More control structures!

Switch / case

Ififififif

```
String inFrench(int number) {
    if(number == 0) {
        return "zéro";
    } else {
        if(number == 1) {
            return "un";
        } else {
            if(number == 2) {
                return "deux";
            } else {
                if(number == 3) {
                    return "trois";
                } else {
                    return "baguette";
                }
            }
        }
```

Slightly better

this is the one valid case when it’s ok to not use `{...}` (in the `else` statement):

```
String inFrench(int number) {
    if(number == 0) {
        return "zéro";
    } else if(number == 1) {
        return "un";
    } else if(number == 2) {
        return "deux";
    } else if(number == 3) {
        return "trois";
    } else {
        return "baguette";
    }
}
```

A new construct!

- a block:

```
switch() {
     ...
}
```

- two inline tests:
  - `case x`: where `x` must be a constant of the same type
  - `default`: catch-all

```
case 0: ...
case 1: ...
default: ...
```

Syntax

assuming

- `e`: an expression of type `t`
- `c1,...,cn` are **constants** of type `t`
- `st1,...,stn`(, `stn+1`) are statements

```
switch(e) {
    case c1: s1;

...

    case cn: sn;
    default: stn+1; // optional
}
```

is a **statement** which

- finds `i` such that `ci == e` (or `n+1`)
- runs **all statements** from `si` to `sn+1`

Example

```
String inFrench(int number) {
    switch(number) {
        case 0: return "zéro";
        case 1: return "un";
        case 2: return "deux";
        case 3: return "trois";
        default: return "baguette";
    }
}
```

`break`

the `case` lines are just *labels*

→ execution jumps and *continues* below from there

- like `return`, stops the current function
- no value, just stops
- actually, works in all loops

Try / catch

Errors happen

- complex program, *dynamic* behaviours
- → hard to anticipate
- → will fail

What to do?

- the program may recover
- exception error messages are “ugly”: users should understand the *cause* if they can do something about it

Syntax

assuming `s0`, `s1` and `s2` are statements

```
try {
    s0
} catch(Exception e) {
    s1
} finally {
    s2
}
```

is a **statement** which

- execute `s0`
- if an exception occurs, execute `s1` (where variable `e` may appear)
- (executes `s2` whether `s0` has failed or not)

More on for

Remember for?

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

Actually. . .

- `<INITIALIZE>` doesn’t have to start at `0`
- `<TEST>` can be any expression, a call to a (`boolean`) function
- `<ITERATE>` can be any statement (you can count up or down, by arithmetic, geometric progression or any function you want)

A beautiful solution

```
int nextCollatz(int i) {
    return i % 2 == 0? i / 2 : 3*i+1;
}
```

A beautiful solution

```java
int nextCollatz(int i) {
    return i % 2 == 0? i / 2 : 3*i+1;
}

for(int i = 221; i > 1; i = nextCollatz(i)) {
    System.out.println(i);
}
```

The end of the loop

do ... while

- variant of `while`
- reverse block execution and test
- $\Longrightarrow$ block is always executed once
- (not conceptually different, just convenient sometimes)

```
int n = 4;
do {
    n++;
} while(n < 2)
```

Now `n` is `5`

continue

- like `break`, stop the loop in the middle
- unlike `break`, will resume the loop normally

```
for(int i = 0; i < 4; i++) {
    if(i % 2 == 0) {
        continue;
    }
    System.out.println(i);
}
```

will print

```
1
3
```

Our first program

Code

A class to represent the program

- in Java a program is made of classes
- represent an instance of the program running (*reflexivity*)
- otherwise *normal*, choose the name you want, can have attributes

```
class Main {
    ...
}
```

A very special function: main

Requirements

- entry point of the program (“where do we start?”)
- no context yet, so can’t instantiate `Main` class
- ⟹ `static`, (`public`)

Types

- can’t return anything (no guaranty it will end): ⟹ output type `void`
- command-line arguments ⟹ input type `String[]`

Example

```
class Main {
    public static void main(String[] arguments) {
        for(String argument : arguments) {
            System.out.println(argument);
        }
    }
}
```

Practice

Compiling

the above saved in a file `Main.java` in current folder.

```
$ ls
Main.java
$ javac Main.java
$ ls
Main.class  Main.java
```

Running it

the `java` command expect a *virtual* path: to the class, not to the file

```
$ java Main one two three four
one
two
three
four
```