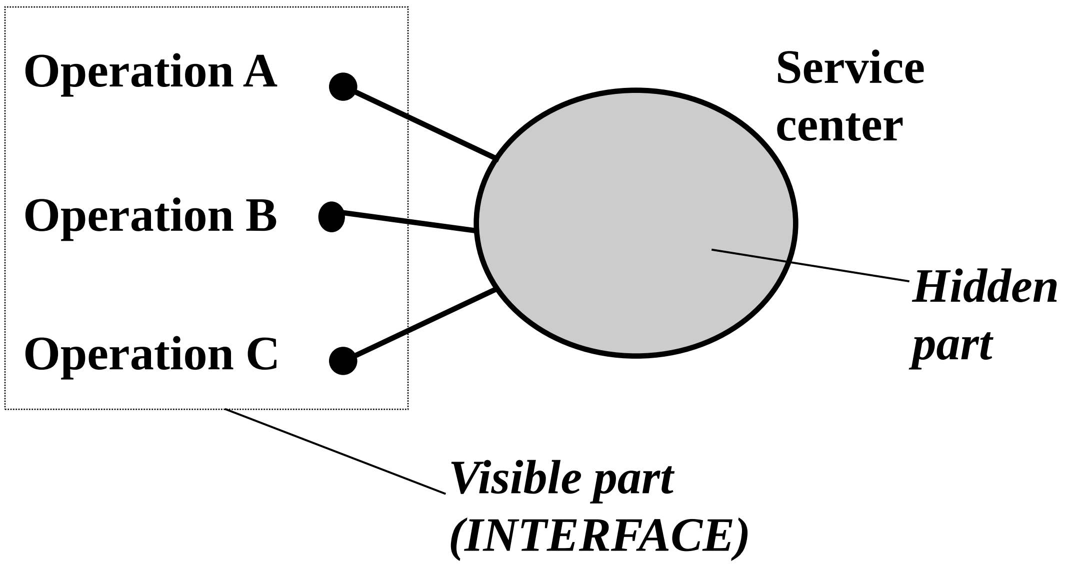
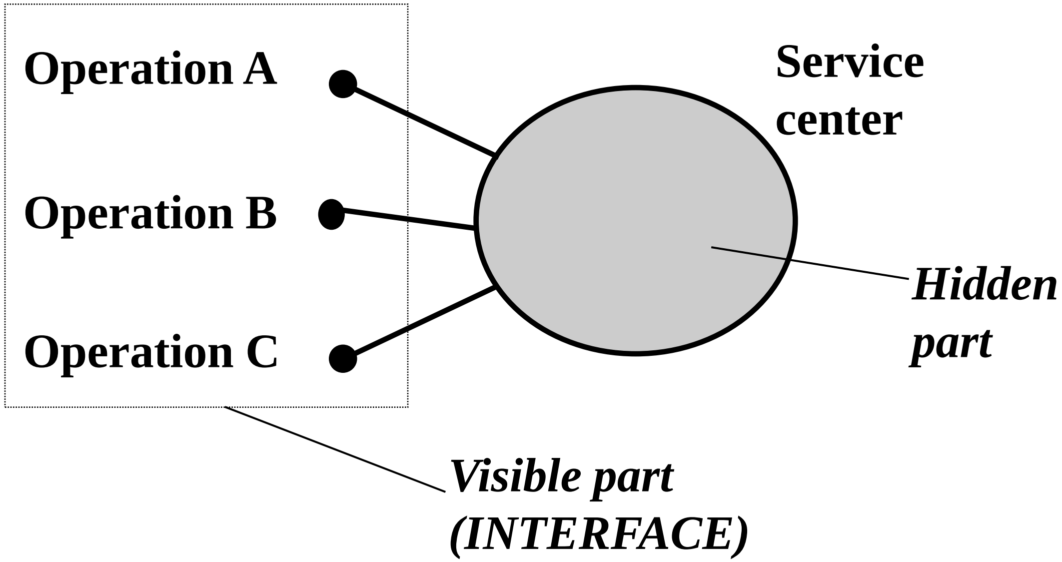
THE CONCEPT OF OBJECT

An object may be defined as a *service center* equipped with a *visible part* (interface) and an *hidden part*



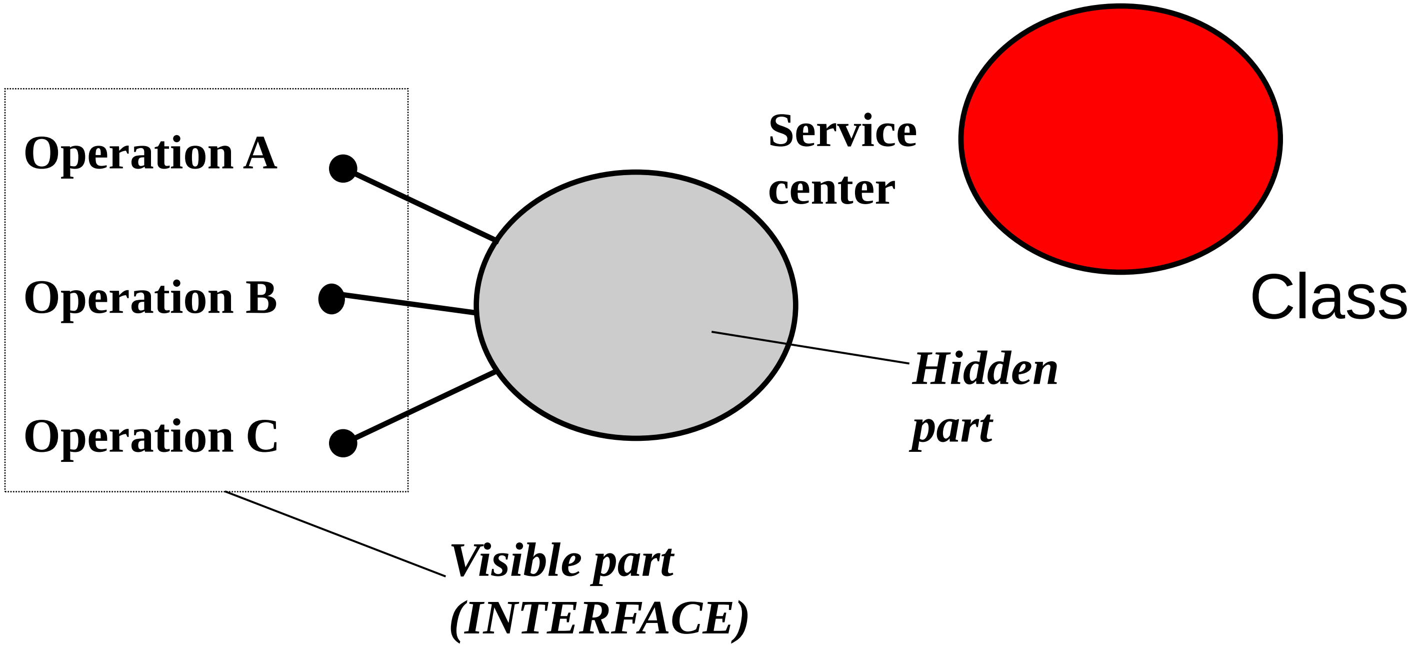
THE CONCEPT OF OBJECT

An object offers to other objects (clients) a set of activities (operations) ***without making known / accessible its internal organization***



THE CONCEPT OF OBJECT

Each client can *create* (*instantiate*) *many objects, as needed, from a sort of "model" of the object* (class)

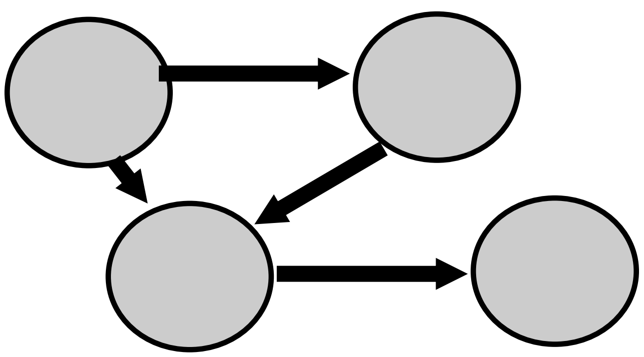


SYSTEMS OF OBJECTS

Architecture of an object-based system:

- **a set of objects that *interact with one each other***
- **without knowing *anything* of their internal representations**
- ***message exchange***

interaction model



OBJECT: BASIC IDEA

- integrates ***data*** and ***elaboration (behavior)***
- promotes both **top-down** and **bottom-up** design and development approaches
- captures the fundamental principles of proper structuring of the software
- introduces very rich interactions oriented to complexity management

OBJECT PROPERTIES

- **An object has *state*, *operation(s)* and *identity***
- **Structure and operation of *similar objects* are defined in their common class of which they are *instances***
- **The terms *instance* and *object* may be used interchangenbly**

THE CONCEPT OF CLASS

- **The *class* describes the *internal structure* and the *behavior* of an obejct**
- **Objects belonging to the same class have:**
  - **the same *internal (state) representation***
  - **the same *operations***
  - **the same *function***

THE CONCEPT OF CLASS

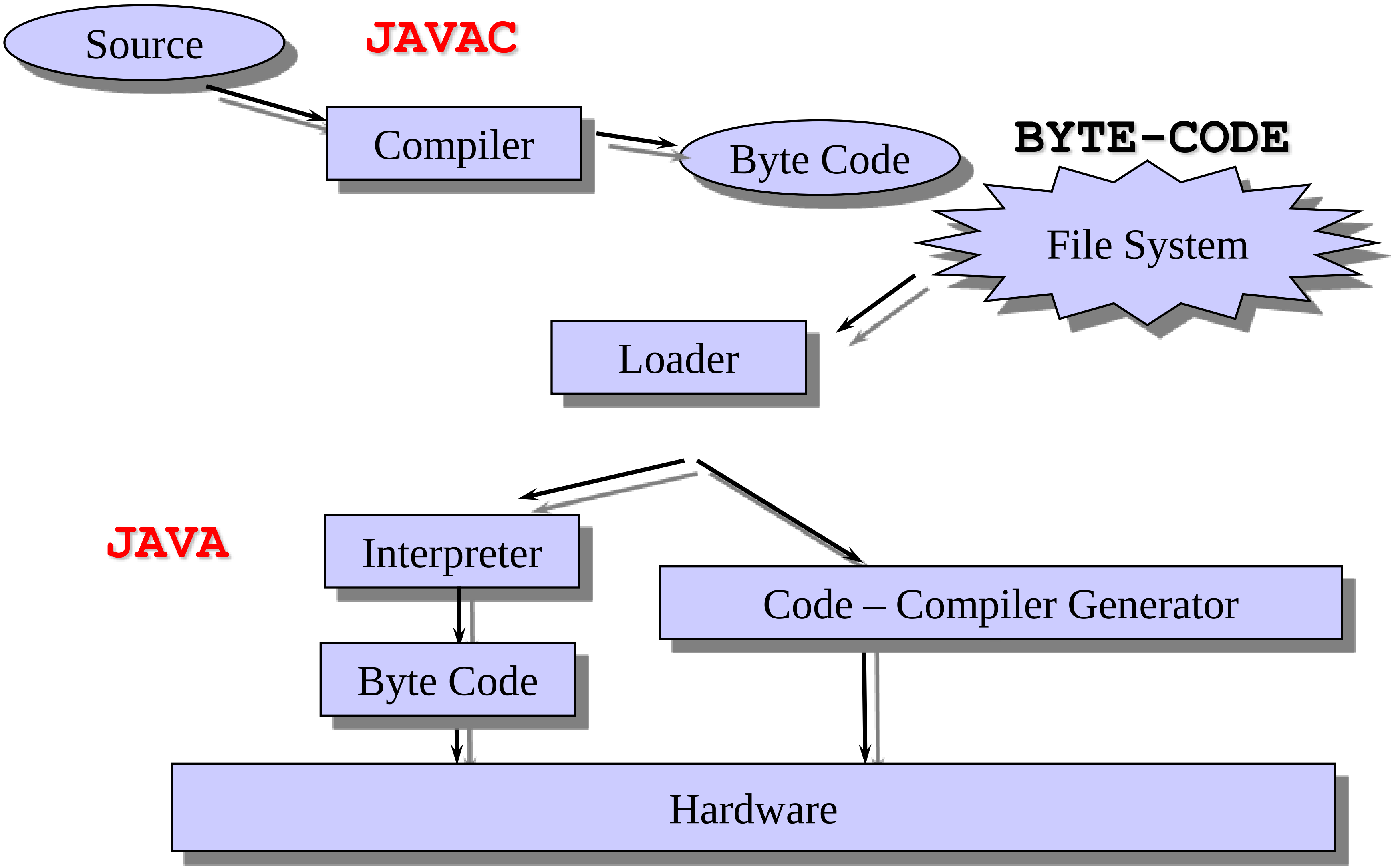
A *CLASS* combines the properties of:

- ***software component:* it can have its *own data / operations***
- ***module:* it encapsulates data and functions, implementing proper *protection mechanisms***
- ***abstract data type:* it acts as a "shape" to *create new objects***

THE JAVA LANGUAGE

- **It is a fully *object-oriented language*: apart primitive types (`int, float, ...`), *there are only classes and objects***
- **It highly inspires to C++, but has been designed *without any backward compatibility requirement w.r.t. C* (even though is is similar…)**
- **A program is a set of *classes***
  - ***even the main is defined inside a class!***

JAVA APPROACH



JAVA CLASSES

A ***Java class*** **is an entity** ***sintactically similar to a*** `struct`

- **but, contains** ***not only data...***
- **… but also** ***functions that operate over those data***
- **And specifies** ***the protection level***
  - ***pubblic*****: visible from other classes**
  - ***private*****: visible only inside the class**
  - **...**

JAVA CLASSES

A *Java class* is an entity with a "*double nature"*:

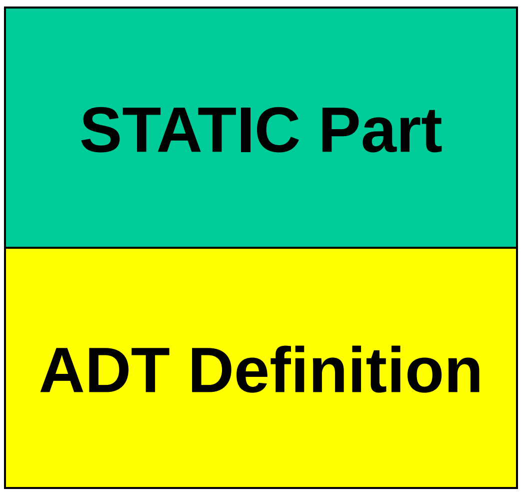
- **it is a *software component,*** that may have its own *data* and *operations,* properly **protected**

- **but it contains also the definition of an *abstract data type,* that is a "shape" to *create new objects,*** that also have proper **protection mechanisms**

JAVA CLASSES

- **The part of a class that realizes the concept of *software component* is called *static part***
  - contains all data and functions that characterize the class as an autonomous software component
- **The other part of the class, that contains the definition of an *Abstract Data Type (ADT) ("schema for objects"),* is the *non-static part***
  - contains data and functions that characterize the objects that will be built *later* using this "schema"

THE CONCEPT OF CLASS



A class is a *software component*: it can have its *data* (STATIC) and its *operations* (STATIC)

A class contains also the *definition of ADT*, usable as a *"blueprint"* to create <u>then</u> new objects (NON static part)

THE CONCEPT OF CLASS

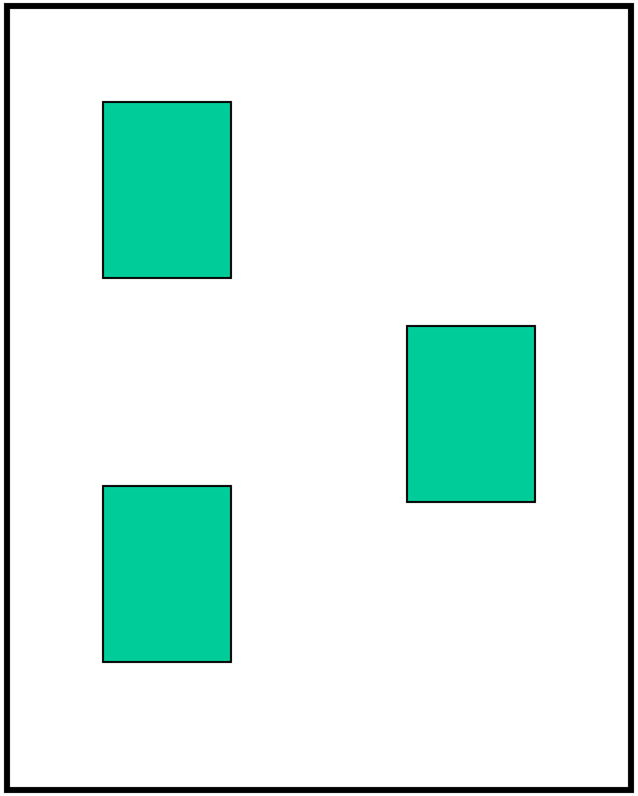
- **If there is only the STATIC part:**
  - **the class operates only as a software component**
  - **it contains data and functions, *as a module***
  - **in addition, it is possible to define appropriate *protection levels***
  - **typical use case: *function libraries***

- **If there is only the NON STATIC part:**
  - **it defines only an ADT**
  - **it specifies the internal structure of a data type, *as structs* (in C)**
  - **in addition, it is possible to specify *also the functions* that operate over those data**

JAVA PROGRAMS

A Java program is *a set of* *classes* *and* *objects*

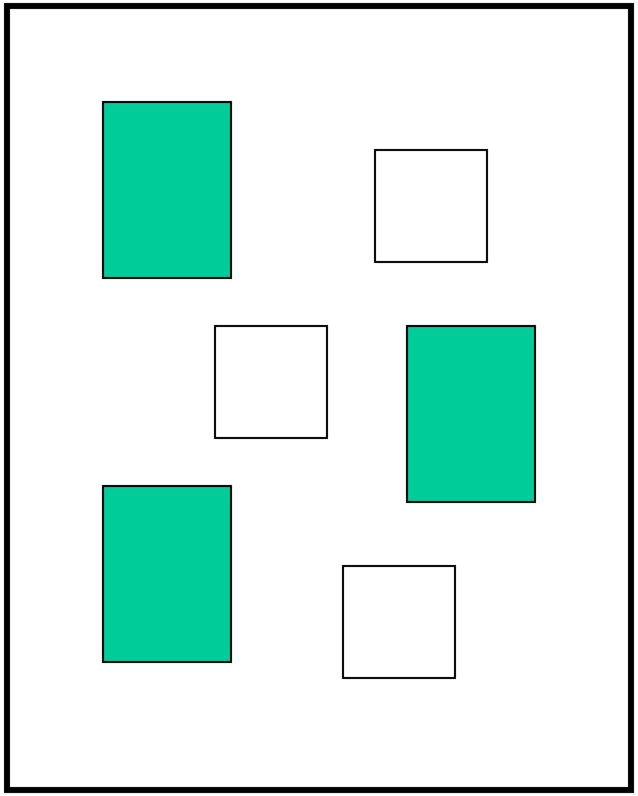
- **The classes are *static* components, that *exist already* at the beginning of the program**



JAVA PROGRAMS

A Java program is ***a set of*** ***classes*** ***and*** ***objects***

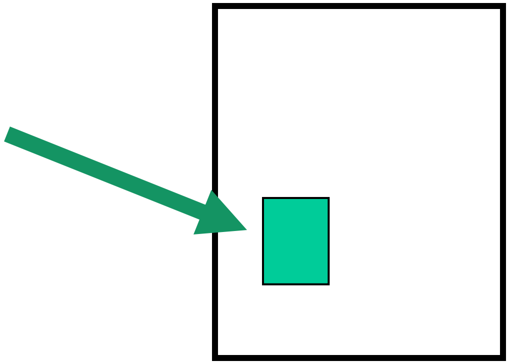
- The **classes** are ***static*** **components, that** ***exist already*** **at the beginning of the program**
- The **objects** **instead are** ***dynamic*** **components, that** ***are dynamically created when needed at runtime***



THE SIMPLEST PROGRAM

- **The simplest Java program is constituted by *a single class* acting as *single software component***

- **It has only the static part**

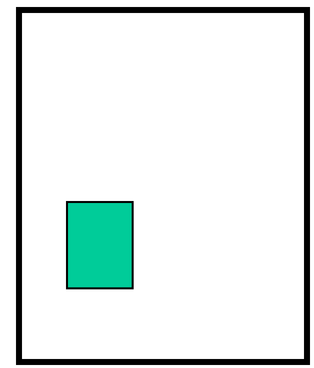


- **At least, it has to define**
  a single (static) function: the `main`

THE MAIN IN JAVA

The main in Java is a *public* function with the following *fixed interface*:

```
public static void
   main(String args[]){
   ......
}
```



- Must be declared **public**, **static**, **void**
- Must not have return value (it is void)
- Must always have line command arguments, *even if they are not used, as a* ***`String`*** *array (the first is not the program name)*

JAVA PRIMITIVE DATA TYPES

- characters
  - `char` (2 byte) UNICODE coding
  - it corresponds to ASCII for the first 127 characters
  - and to ANSI / ASCII for the first 255 characters
  - *char constants also expressed as* `'\u2122'`
- integers (signed)
  - `byte` (1 byte) -128 ... +127
  - `short` (2 byte) -32768 ... +32767
  - `int` (4 byte) -2.147.483.648 ... 2.147.483.647
  - `long` (8 byte) $-9\ 10^{18}$ ... $+9\ 10^{18}$

NB: long constants end with the letter L

JAVA PRIMITIVE DATA TYPES

- **real *(IEEE-754)***
  - **`float` (4 byte)** **$- 10^{45}$ ... $+ 10^{38}$**
    (6-7 significant digits)
  - **`double` (8 byte)** **$- 10^{328}$ ... $+ 10^{308}$**
    (14-15 significant digits)
- **boolean**
  - **`boolean` (1 bit)** **`false` e `true`**
  - **independent type *totally decoupled from integers:* it is not possible to turn boolean into integers and viceversa, *not even with a cast***
  - **all relational and logical expressions return as a result a `boolean`, and no more an `int` (as it was in C)!**

OBJECTS
EXAMPLE: THE COUNTER

- **<u>This class does not contain *its own* data or functions *(static)*</u>**
- **It supplies only the definition of an ADT** that will be used then to instantiate objects

```
public class Counter {
  private int val;
  public void reset() { val = 0; }
  public void inc()   { val++; }
  public int getValue() {
       return val;
  }
}
```

Data

Operations (behavior)

Unique linguistic construct for data and operations

OBJECTS
EX… THE COUNTER

- **This …a or funct…**
- **It sup…** … that will be…

The field `val` is *private*: **it can be accessed *only by operations defined within the same class* (reset, inc, getValue), *and not by any other!***
It grants encapsulation

```java
public class Counter {
  private int val;
  public void reset() { val = 0; }
  public void inc()   { val++; }
  public int getValue() {
      return val;
  }
}
```

Data

Operations (behavior)

Unique linguistic construct for data and operations

JAVA OBJECTS

- **The OBJECTS are "dynamic" components:** ***are created "on-the-fly"*, when they are used, through the `new` operator**
- **They are created *as an copy and similar to a class*** *(non-static part),* **that describes its *properties***
- **Over them, it is possible to invoke *the public operations* exposed by the class**
- **It is not needed to take care of object destruction: Java has a *garbage collector!***

OBJECT CREATION

To create an object:

- **<u>first</u> a *reference* is defined, its type is *the name of the class that acts as model***
- ***<u>then</u> it creates dynamically the object*** **through *the operator `new`*** *(similar to C malloc)*

Example:

```
Counter c;                  // reference definition
...
c = new Counter();          // object creation
```

JAVA OBJECTS

Use: *"message passing" style*

- **not a function with the object as parameter…**
- **…but rather *an object over which <u>methods</u> are invoked***

For instance, if `c` is a `Counter`, a client can write:

```
c.reset();
c.inc(); c.inc();
int x = c.getValue();
```

COMPLETE EXAMPLE

```
public class Example1 {
 public static void main(String v[]) {
    Counter c = new Counter();
    c.reset();
    c.inc(); c.inc();
    System.out.println(c.getValue());
 }
}
```

- **The main creates a new object Counter…**
- **...and then uses it by *name*, with *dot notation…***
- **…*without the need to dereference it explicitly!***

EXAMPLE: DEVELOPMENT

- **The two classes <u>must</u> be written *in two separate files*,** called, respectively:
  - **`Example1.java` (it contains the class `Example1`)**
  - **`Counter.java` (it contains the class `Counter`)**
- **That is necessary because both classes are *<u>public</u>*: in a `.java` file there can be *one only <u>public</u> class***
  - ***but there can be other, non public, ones***
- **To compile:**

Note: the order does not matter

```
javac Example1.java Counter.java
```

EXAMPLE: DEVELOPMENT

- **The two classes <u>must</u> be written *in two separate files*,** called, respectively:
  - **`Example1.java` (contiene la classe `Esempio1`)**
  - **er)**

- **s**
  one

Also separately, but *<u>in order</u>*:

```
javac Counter.java
javac Esempio1.java
```

The class `Counter` must <u>already exist</u> when the class `Esempio1` is compiled

- **To compile:**

```
javac Example1.java Counter.java
```

EXAMPLE: EXECUTION

- **The compilation of those two files generates *two files* `.class`**, called, respectively:
  - **`Example1.class`**
  - **`Counter.class`**
- **To run the program it is sufficient to invoke the interpreter (`java`) with the *name of the (public) class* that contains the main**

```
java Example1
```

ERROR MANAGEMENT

- **Often there are “critical” instructions, that under certain conditions may produce errors**
- **The classical approach consists in *inserting controls* (if… else..) *trying to a priori intercept* critical situations**
- **But this management way is ofter *unsatisfactory***
  - **it is not easy to foresee all the situations that may produce errors**
  - **“managing” the error ofter means only to print a message on the screen**

EXCEPTIONS

Java introduces the concept of *exception*

- **Instead of *trying to foresee* error situations, it *tries to execute* the operation *in a controlled code block***
- **if the error situation occurs, the operation *raises an exception***
- **the exception is *caught* by the code block where the operation has been executed…**
- **… and can be *managed* in the most appropriate way**

EXCEPTIONS

```
try {
   /* critical operation that may
      raise exceptions */
}
catch (Exception e) {
   /* exception management */
}
```

If the operation raises *different types* of exceptions in response to different types of error, *more* ***`catch`*** *blocks* may follow the same ***`try`*** block

WHAT IS AN EXCEPTION in JAVA

- **An exception *is an object*, instance of `java.lang.Throwable` or one of its subclasses.**
- The two most common subclasses are `java.lang.Exception` and `java.lang.Error`
- The word “exception”, however, often refers to both of them

WHAT IS AN EXCEPTION

- **An `Error` indicates problems related to class loading and function of the Java virtual machine (es. not enough memory), and is considered not *recoverable*:**
  hence *it should be not* caught
- **An `Exception`, instead, indicates *recoverable* situations, at least in principle** (end of file, array index out of bounds, input errors, etc.):
  it should be *caught* and *managed*

JAVA ARRAY

- **Java arrays are *objects,* instances of a *special class* defined by `[ ]`**
- **Hence, <u>the *reference* is *defined* (as for any object)...</u>**

```
int[] v;              int v[];
Counter[] w;          Counter w[];
```

- **...and then <u>the *object* is *dynamically created*:</u>**

```
v = new int[3];
w = new Counter[8];
```

JAVA ARRAY

- Java arrays are *objects,* instances of a *special*

> **It is a reference, hence *it does not have to specify any dimension!***

- ...ct)...

```
int[] v;            int v[];
Counter[] w;        Counter w[];
```

> The position of `[]` is either after the name, as in C, or after the type ***(not available in C)***

- ...:

```
w = new Counter[8];
```

> **The dimension *is specified at the creation (“`new`” execution)***

JAVA ARRAY

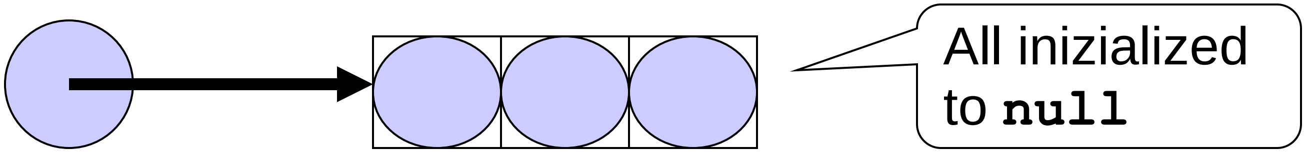
Attention!! **Each array element:**

- ***is a variable*, if the array elements are *of primitive type* (`int`, `float`, `char`, ...)**

```
v = new int[3];
```

- ***is a reference to a (future) object*, if the array elements are *(references to) objects***

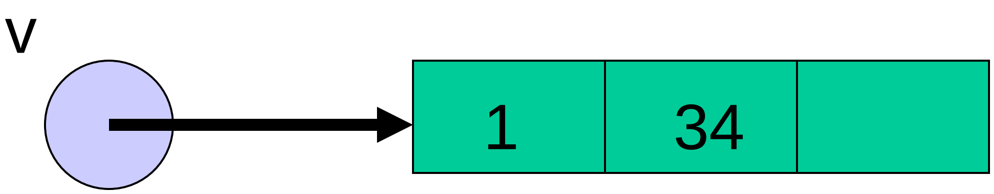
```
w = new Counter[3];
```



JAVA ARRAY

Hence, in the first case, *primitive value arrays*, each array element is a normal variable, "already usable" as is:

```
v = new int[3];
v[0] = 1; v[1] = 34;
```



APPENDIX:

STRUCTURED PROGRAMMING

STRUCTURED PROGRAMMING

- **<u>Goal:</u> make easier to read programs (hence also their modification and maintenance).**

- **Suppression of unconditional jumps (*go to*) in the control flow.**

- **The executive part of a program can be seen as a (complex) command obtained from the *elementary instructions*, using certain rules of composition (*control structures*).**

CONTROL STRUCTURES

Key concepts:

- **concatenation and composition** CODE BLOCK
- **conditional instruction** SELECTION
  - branches the control flow based on the true/false value of a boolean expression ("*choice condition*")
- **repetition and iteration** CICLE
  - executes repetitively an instruction until a certain boolean expression is true ("*iteration condition*")

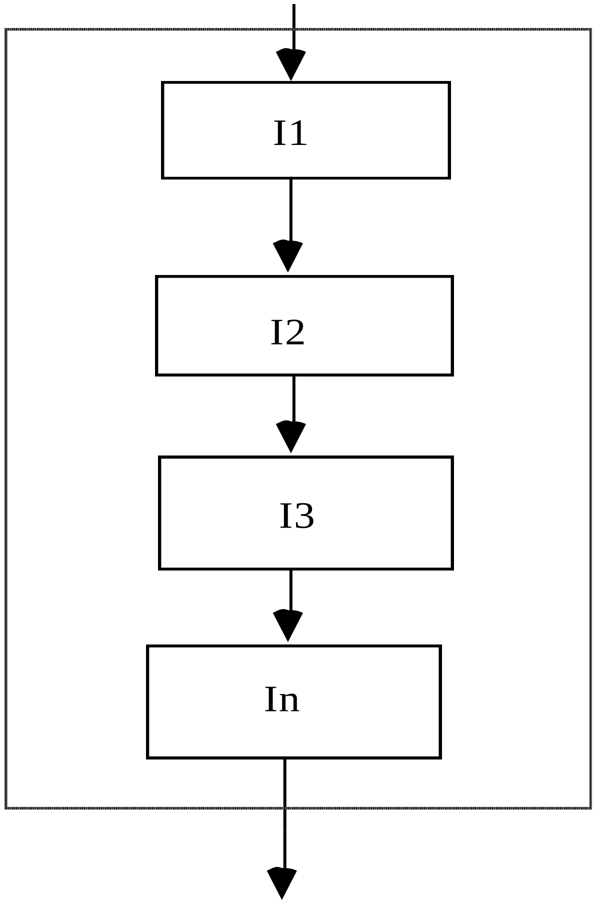
(CODE) BLOCK

<block> ::= {

[<statements and definitions>]

{ <instructions> }

}



- The visibility scope of block symbols is restricted to the block itself
- after a block the semicolon is not needed (but it *terminates* simple instructions)

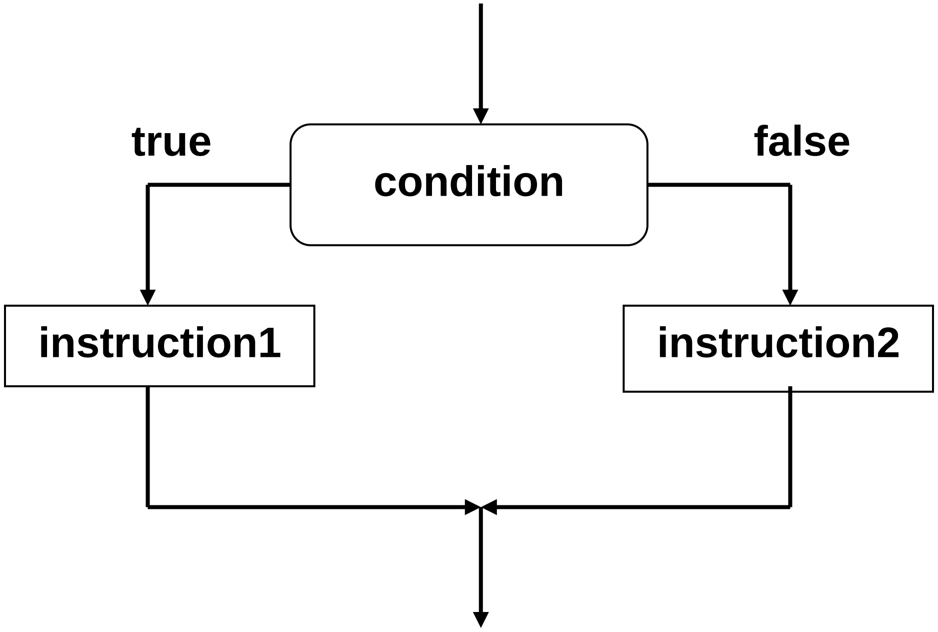
CONDITIONAL INSTRUCTIONS

```
<selection> ::=
        <choice> | <multiple-choice>
```

- the second *is not essential*, and we will not see it.

SIMPLE CHOICE INSTRUCTION

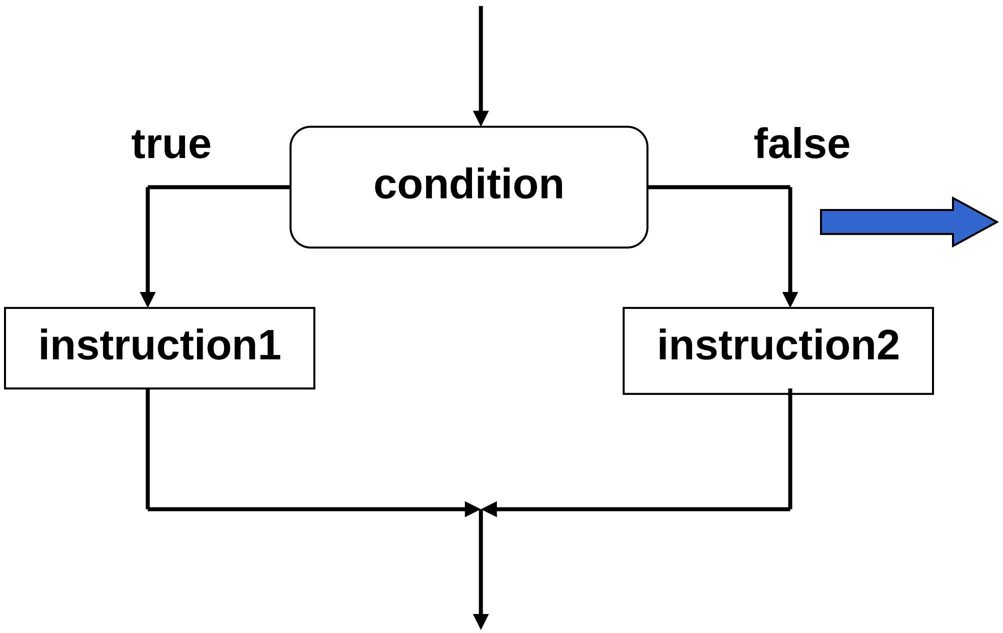
```
<choice> ::=  if (<cond>) <instruction1>
              [ else <instruction2> ]
```



The condition is evaluated when the “if” is executed.

SIMPLE CHOICE INSTRUCTION

```
<choice> ::=  if (<cond>) <instruction1>
              [ else <instruction2> ]
```



The **`else`** part is *optional*:
if omitted, when the condition is false the control flow continues with the instruction that follows the **`if`**

EXAMPLE of `if` INSTRUCTION

- <instruction1> and <instruction2> are, each one, *single instructions*
- If it is necessary to specify more instructions, it is necessary to use a *block*

```
if (n > 0) {            /* block beginning */
  a = b + 5;
  c = a;
}                       /* block end */
  else n = b;
```

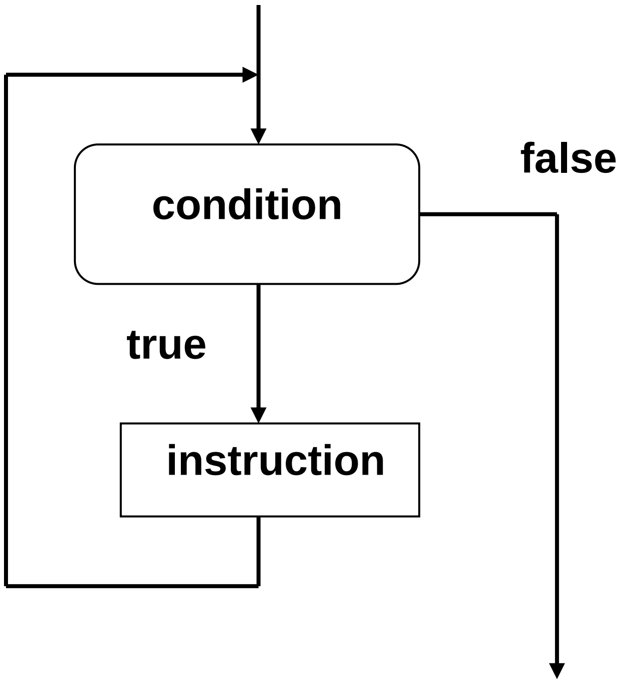
ITERATION INSTRUCTIONS

```
<iteration> ::=
        <while> | <for> | <do-while>
```

- Iteration instructions:
  - have *one only entry point* and *one only exit point* in the program flow
  - hence, they can be interpreted *as a single action* in sequential computation

`while` INSTRUCTION

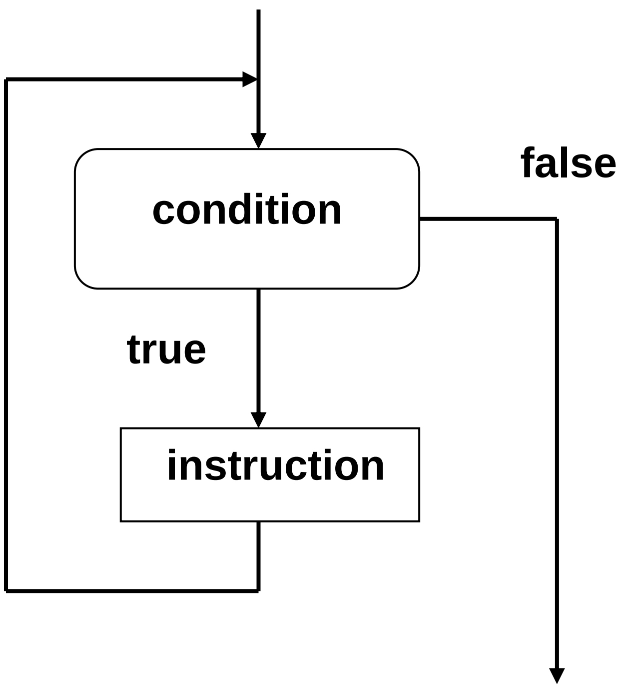
```
<while> ::=
    while(<condition>) <instruction>
```



- The instruction is repeated *until the condition is/remains true*
  - If the condition is false, the iteration is not executed (*not even one time)*
  - In general, *it is not known (in advance)* ***how many times*** the instruction will be repeated

`while` INSTRUCTION

```
<while> ::=
    while(<condition>) <instruction>
```



Before or afterwards, *directly or indirectly*, the instruction has to *modify the condition:* otherwise, the iteration will last *forever!*

INFINITE CYCLE

Hence, typically the *instruction is a block,* within which some of the *variables that appear in the condition is modified* (to avoid infinite cycling)

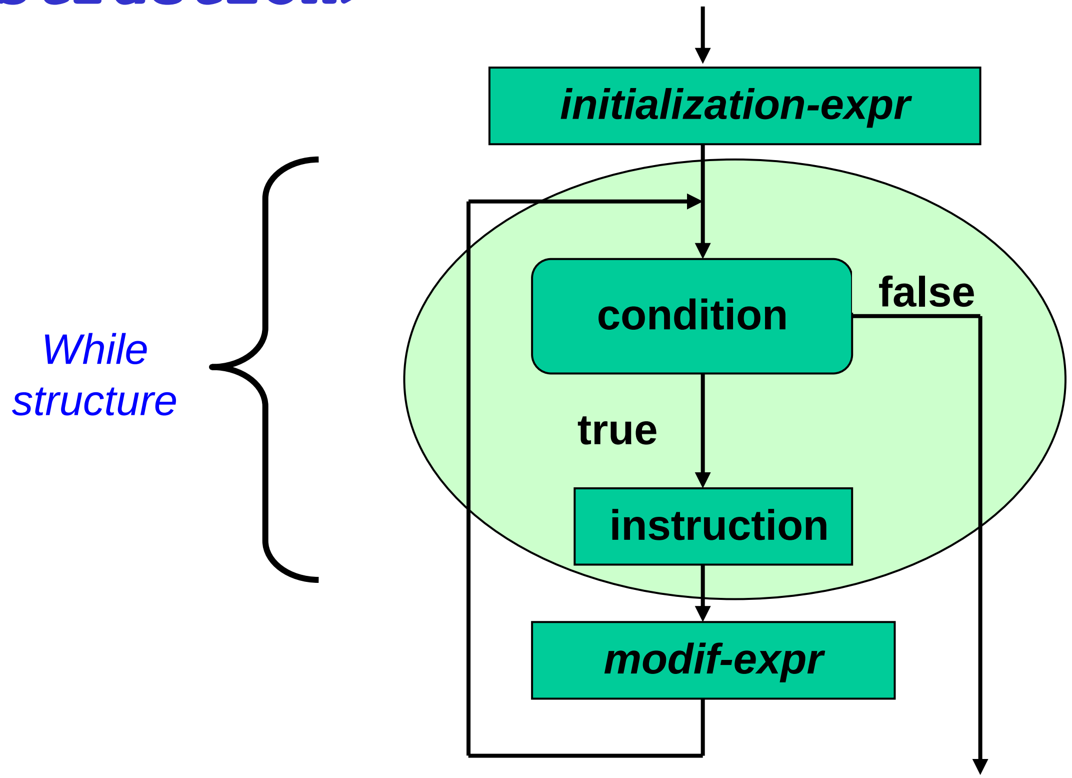
`for` INSTRUCTION

- It is an evolution of the `while` instruction aimed to avoid some frequent mistakes:
  - lack of *variable initialization*
  - lack of *variable modification phase within the cycle* (endless cycle loop risk)

- In general, it is used when it is well-known how many times the cycle has to be executed.

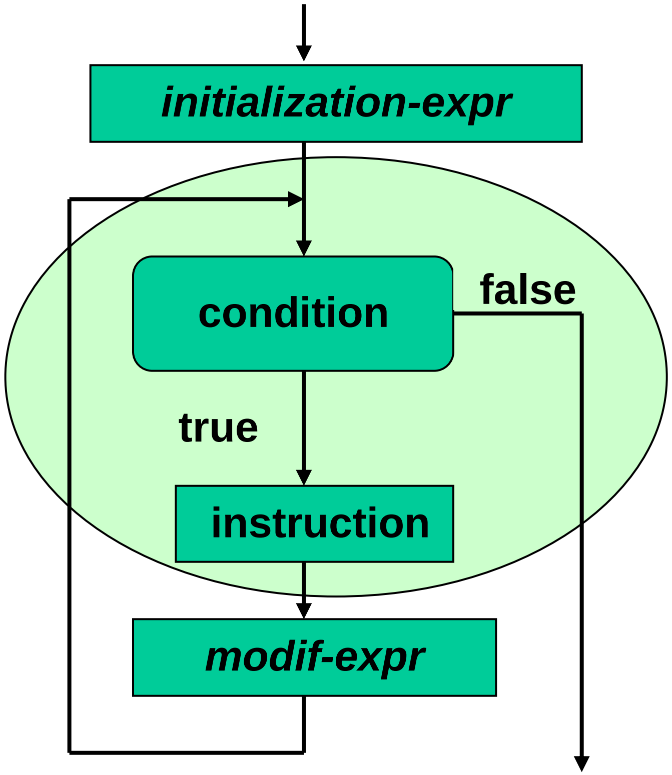
`for` INSTRUCTION

```
<for> ::=
  for( <init-expr>;<cond>;<modif-expr>)
  <instruction>
```



`for` INSTRUCTION

```
<for> ::=
  for( <init-expr>;<cond>;<modif-expr>)
  <instruction>
```



Initialization expression:

`<init-expr>`

evaluated *one and one only time* *__before__* the iteration begins.

Condition: **`<cond>`**

evaluated *for each iteration*, to decide if prosecuting (as in while).
If missing it is assumed *__true by default__*!

Modification expression: **`<modif-expr>`**

evaluated *for each iteration*, *__after__* the instruction has been executed.