Patrick GIRARD
University of Poitiers
2009-2012©

Object-Oriented Programming in Java™

What is object-oriented programming ?

Chapter 1

Table of contents

Ways of programming

- Imperative programming
- Functional programming
- Logical programming
- Object-Oriented Programming

Assembly language

- Manipulated objects
  - Register, memory location
- Main features
  - If-GOTO for expressing the control
  - No type
  - Very low level
  - Processor dependent

FORTRAN

- FORmula TRANslator
- Manipulated objects
  - numbers, arrays
- Main features
  - static data
  - no type creation
  - But… it allowed to send people on the Moon…
- Main drawback
  - not dynamic

FORTRAN: example

```
      INTEGER FUNCTION PPCM(A,B)
      INTEGER A,B,R,Q,X,Y
      X=A
      Y=B
10    Q=X/Y
      R=X-Q*Y
      IF(R.EQ.0) GO TO 20
      X=Y
      Y=R
      GO TO 10
20    PPCM=A*B/Y
      RETURN
      END
```


COBOL

- COmmon Business Oriented Language
- Manipulated objects
  - numbers, records
- Main features
  - files
  - few usage of arrays
- Main drawback
  - very verbose

COBOL: example

```
IDENTIFICATION DIVISION.
PROGRAM-ID PROG.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
…
FILE SECTION.
FD PERSONNEL RECORD LABEL STANDARD
       DATA RECORD ARE EMPLOYE PRIME-RETENUE.
01 EMPLOYE.
    05 ECODE   PIC 99.
    05 ENOM    PIC X(20).
    05 ENUMSS  PIC X(13).
    05 ECHELON PIC 9(2).
…
PROCEDURE DIVISION
10. OPEN INPUT FPERSONNEL.
    READ FPERSONNEL AT END GO TO 130.
20. OPEN OUTPUT FPAIE.
…
```

Structured Programming

- Pascal / C
- Manipulated objects
  - new types, arrays, records
- Main features
  - Hierarchical subroutines (procedures, functions)
  - Pointers and dynamic programming
- Main drawback
  - Not easy when programs grow up

Modular Programming

- Modula
- Main goal
  - "Programming in the large"
- Main features
  - Module / Package
  - Data Hiding / Encapsulation
  - Reliability (strong typing, exceptions)
- Main drawback
  - Replaced by a more evolved idea: object-orientation...

Ways of programming

- Imperative programming
- Functional programming
- Logical programming
- Object-Oriented Programming

Functional Programming

- LISP (1958) - LISt Processing
- Main goal
  - Applying Lambda Calcul to list processing
  - Expression-driven language
  - Mainly used in Artificial Intelligence
- Main features
  - Lists of lists
  - No affectation

Functional programming

- Example : the square function

```
(* the square function *)
let square x = x*x ;;
-: val square : int -> int = <fun>

square 3  ;;
- : int = 9
```


Ways of programming

- Imperative programming
- Functional programming
- Logical programming
- Object-Oriented Programming

Logical Programming

- PROLOG (1972 - France)
  - "PROgrammer en LOGique"
- Main goal
  - Describing the "What", not the "How"
  - Mainly used in Artificial Intelligence
- Main features
  - Writing "rules"

Logical programming

- Explanation:
  - impaire(3)
    - = ! impaire(2)
    - = ! (! impaire(1))
    - = (! (! true))
    - = ! false = true

```
impaire(1).
impaire(X+1) = ! impaire(X).

impaire(3) ?
:- true.
```

One question

How Ada
might be classified?

Ways of programming

- Imperative programming
- Functional programming
- Logical programming
- Object-Oriented Programming

Object-Oriented Programming

Imperative programming
Functional programming
Logical programming
Object-Oriented Programming

Object-Oriented Programming

Table of contents

Table of contents

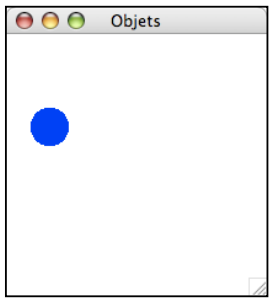
Thinking objects, a natural way

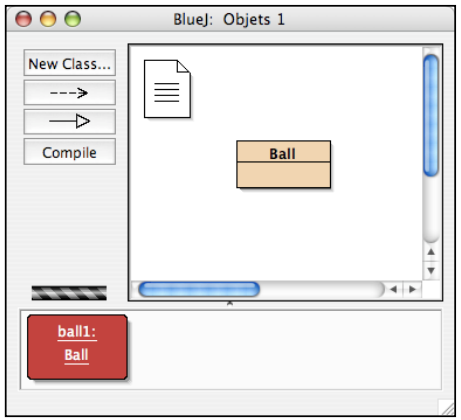
- A "car", a "student", a "ball"
- Objects own characteristics
  - The mileage of a car
  - The colour of a car
  - The name of a student
  - The colour of a ball
  - The size of a ball

Attributes

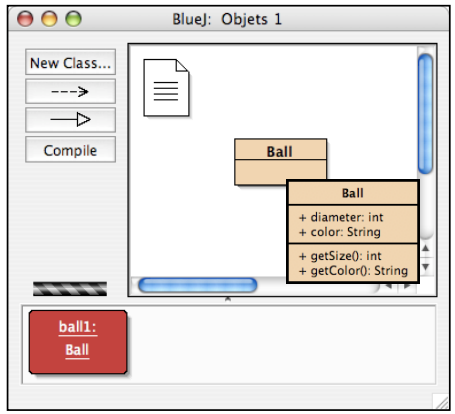
Example

- A ball





Demonstration



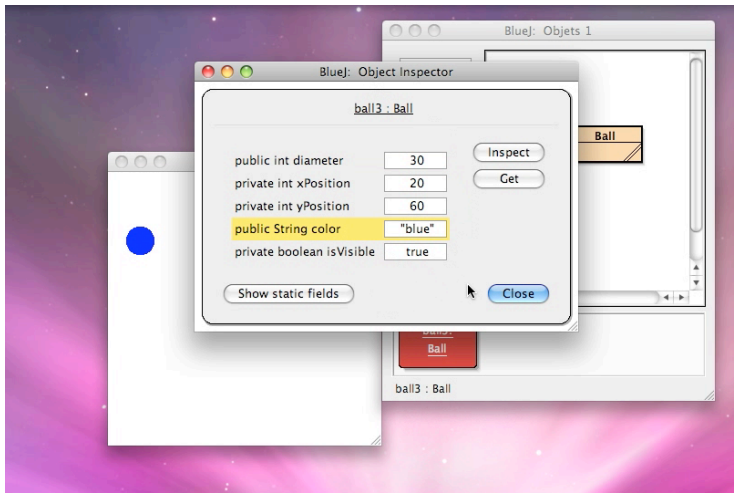
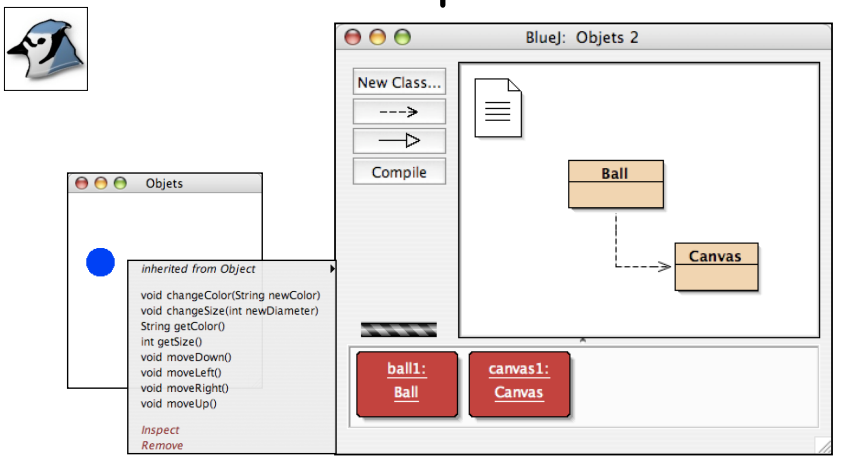


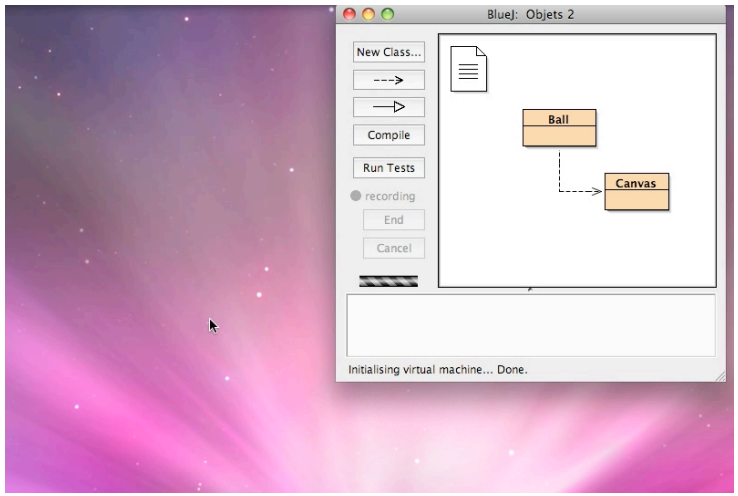
Table of contents

Objects communicate (1)

- Objects can receive messages
  - The ball receives the message "Move Right"
  - The ball receives the message "ChangeColor"

Example





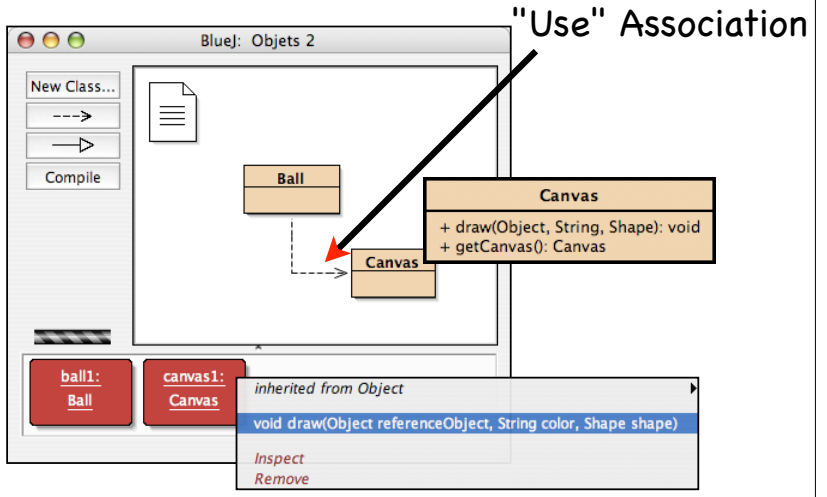
Objects communicate (2)

- Objects can receive messages
  - The ball receives the message "Move Right"
  - The ball receives the message "ChangeColor"
- Objects can send messages
  - The driver starts the car
    - The driver object sends the message "start" to the car
  - The ball sends the message "drawObject" to the canvas

Objects communicate (3)

- Objects know objects
  - To send a message, the sender needs to know the receiving object. Sender and receiver are associated
- Example
  - The ball sends the message "drawObject" to the canvas
  - The ball must know a way to identify the canvas (its name)

Illustration

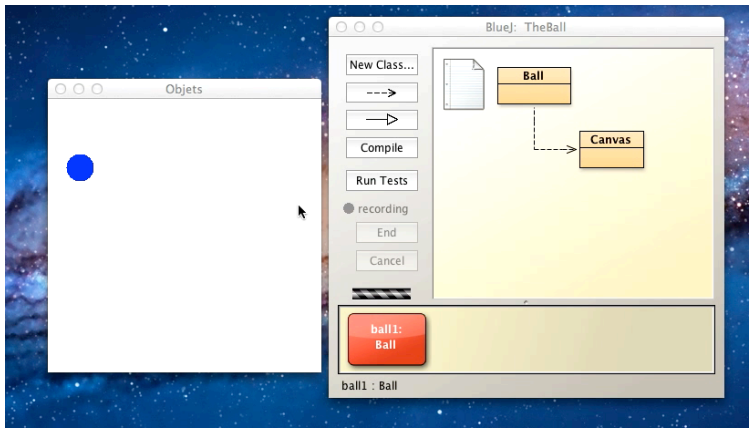


Message typology (1)

- Accessors
  - Messages that deliver information about objects
    - They return a value to the sender
    - They do not modify the receiving object
  - Example:
    - The ball is able to deliver its size

getSize(): int ← return type

UML Notation



Message typology (2)

- Modifiers/Mutators
  - Messages that change the object
    - They do not return any value to the sender
    - They do modify the receiving object
  - Examples:
    - The ball is able to change its size
    - The ball is able to move

changeSize(): void moveRight(): void

Notice: void

Naming conventions (1)

- Accessors
  - get + name of the characteristics
  - is + name if boolean
- Examples:
  - returns the size of the ball
  - returns the status (visible/not visible, true or false) of the ball

getSize(): int isVisible(): boolean

Naming conventions (2)

- Modifiers/Mutators
  - set + name of the characteristics
  - change + name of the characteristics
  - or any meaningful word
- Examples:

setSize(int newSize): void

changeSize(int newSize): void

moveRight(): void

first letter in lower case

then, capital letters

Example: a heater

- A heater is an object that emits heat or causes another body to achieve a higher temperature. Heaters exist for all states of matter, including solids, liquids and gases.
- The opposite of a heater is an air cooler (for air conditionning)

Basic analysis

- The heater manages the temperature
  - An attribute, currentTemperature, is required
- Basically speaking, we need:
  - An accessor, getTemperature(): int, to access the temperature
  - A modifier, setTemperature(int newTemp): void, to change the attribute

A naive question

- Why do we need an accessor and a modifier?
- Why don't we directly access and modify the attribute?

Table of contents

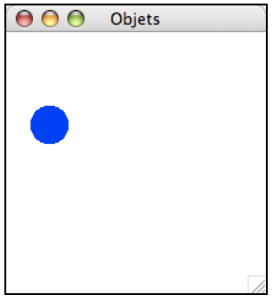
Encapsulation, a good idea

- Do you need to know...
  - How a car engine really works to drive a car?
  - How Internet really works to browse?
  - How a cooking recipe is to eat a meal?
- Obviously not!
  - It's the same for objects

Objects manage attributes

- A "ball" is defined by
  - a size
  - a colour

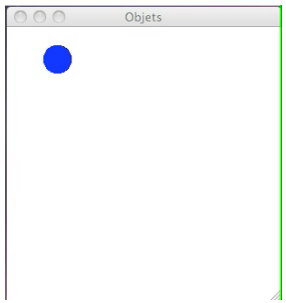
Obvious attributes



Objects manage attributes

- Is it sufficient for the whole ball's behaviour?
- In fact, we can...
  - move the ball
  - hide/show the ball

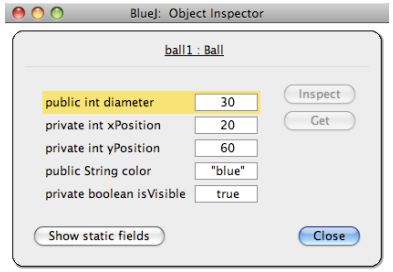
Need for hidden attributes



Objects manage attributes

- A "ball" is also defined by
  - a position in the space (xPosition, yPosition)
  - a state attribute to record the fact that the object is shown or hidden (isVisible)

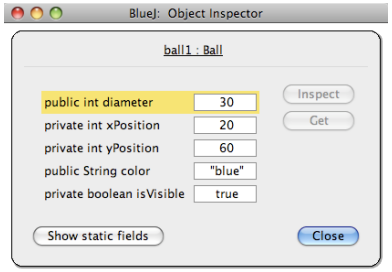
Hidden attributes



Objects manage attributes

- Why "hidden attributes"?
  - Logical actions are not always linked to simple actions on attributes
  - Actions may not be natural on attributes
  - Modifying attributes requires the object to do actions

private attributes



Task-Oriented Messages

- Messages, and specifically modifiers, are often not directly relied to a single attribute
  - For example, moveDiag(), for moving diagonally, would rely to X and Y parameters...
- Message that is nor an accessor nor a modifier... = task-oriented messages

Calculated attributes

- Attributes do not exist in all cases...
  - For example, the "surface" of a circle is an attribute (a characteristics), but the object may use only calculation on demand...
  - So, it can be accessibe only through an accessor, getSurface(): int, without a concrete attribute, which calculates it from the diameter.

The heater (cont.)

- A better solution
  - Generally, a heater is controlled through a button that allows increasing or decreasing the temperature.
  - To be task-oriented, a better solution might be...
    - increaseTemperature(): void
    - decreaseTemperature(): void

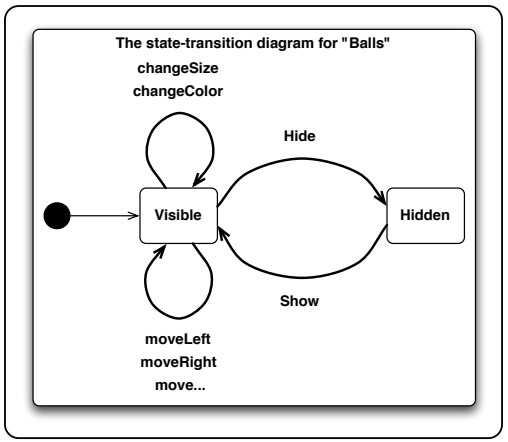
Objects manage their state

- Attributes define a state
  - Visible / Hidden
- Action may not always be possible

State	Visible	Hidden
Allowed actions	ChangeSize ChangeColour Move... Hide	Show

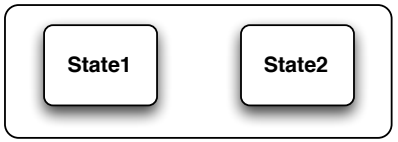
Objects manage their state

- State-Transition diagrams, a good way to model object behaviour

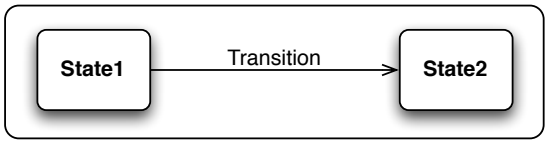


The state-transition diagram

- States



- Transitions

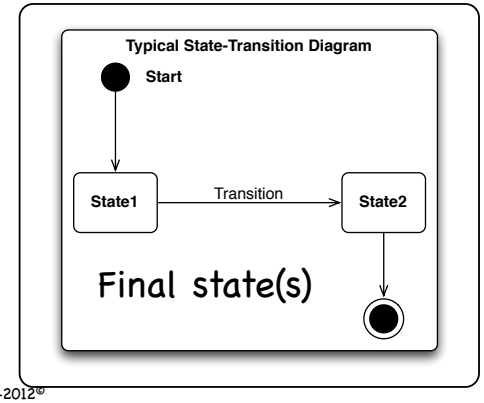


The state-transition diagram

Initial state

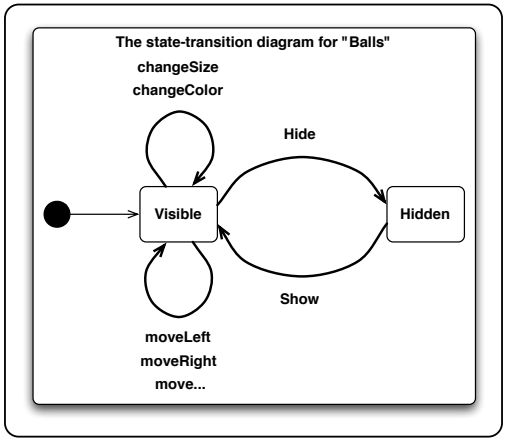
- Initial and final states

Optional



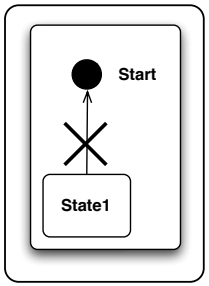
Description of the Balls behaviour

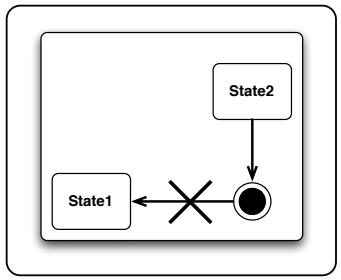
- Only two states
- An initial state with an automatic transition
- One "active" state
- One "passive" state
- No final state
- Transitions are actions



The state-transition diagram

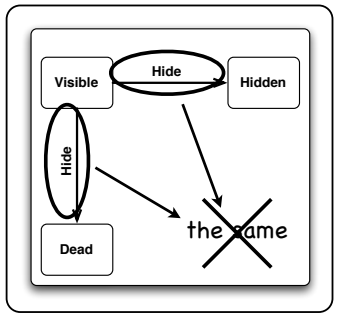
- Some rules (light version)
  - Initial state:
    - no entering transition
    - 0 to 1
  - Final state:
    - no starting transition
    - 0 to n





The state-transition diagram

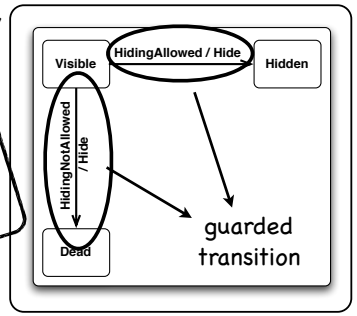
- Some rules (light version)
  - STD are deterministic
    - The same transition cannot start from the same state



The state-transition diagram

- Some rules (light version)
  - Conditions can be added to fire the transition
    - Assume we want a behaviour such as:
      - if Hiding is not allowed, the objects dies !
  - They still are deterministic

More details on STD ... later on



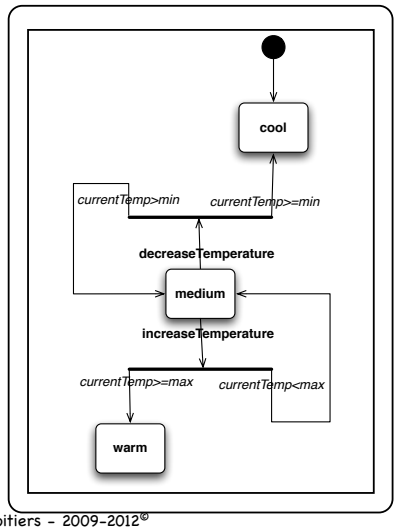
The heater (cont.)

- Depending on needs for safety, a heater cannot be warmer than a specific limit (max), nor cooler than another limit (min).
- There are three states, (1) cool, (2) medium, and (3) warm.

The heater (cont.)

- When in "medium" state, both task-oriented actions are possible
- Depending on the result of the action, the resulting state may be different

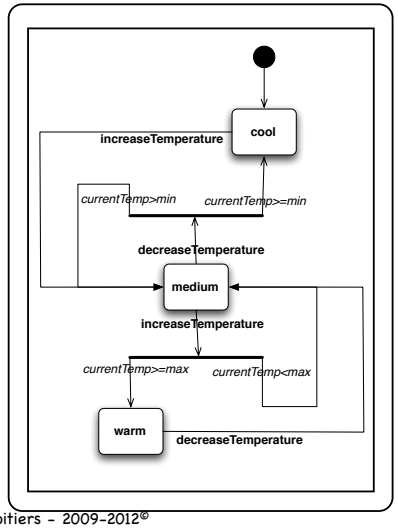
The heater: StateChart (1)



The heater (cont.)

- When in "cool" state, decreaseTemperature() must not be allowed
- When in "warm" state, it's the opposite

The heater: StateChart (2)



The heater (cont.)

- The remaining action are always possible, but do not change the state

The heater: StateChart (3)

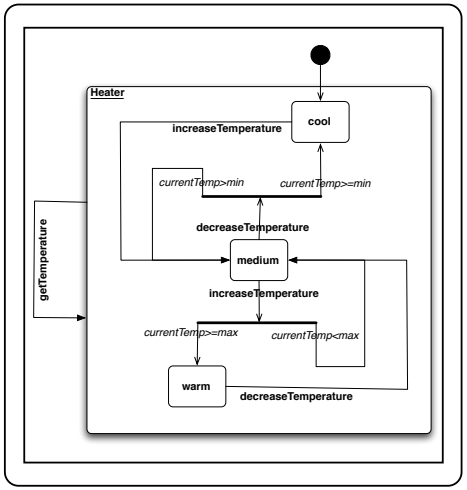


Table of contents

Objects ≠ Classes

- Objects model real objects from a problem domain
  - The black Renault Espace, registred as 1380 TA 86, is my car
- Classes categorize objects
  - Cars synthetize a category of objects. It is a class

Objects ≠ Classes

- Classes describe a kind of object
  - They look like "types" you already know
  - They are "abstract definitions" of objects
- Objects are the individuals
  - They are built from classes (one in fact)
  - They are "concrete objects"
  - They have a "life cycle"

Object creation

- In classical programming...
  - Variables are "declared"...

```
i: integer;      -- Ada
int i;       /* C language */
```

  - ... that's all. Nothing else to do !

Object creation

- In object-oriented programming...
  - Objects must be "declared"...

```
String s;      // Java
Circle c;      // C++
```

  - ... that's not all ! They must be created

```
s = new ("Hello")      // Java
c = new Circle(10);    // C++
```


Constructors

- Sometimes, actions must be made to allow the object to be correct
  - A mail server must create a list of messages
  - A mail client must know the Mail Server to be able to send it messages, and must have a name

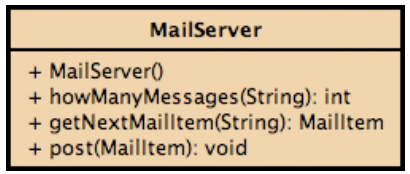
```
MailServer myServer = new MailServer();
MailClient myClient = new MailClient (myServer, "Jones");
```

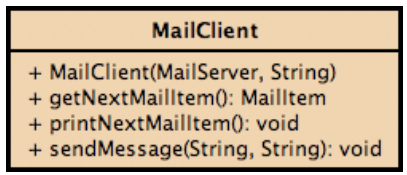

The heater (cont.)

- For the heater, we need to give a start value for the temperature... in the constructor
- We could also specify an increment for the increase/decrease functions... by the constructor
  - Heater(int increment)

What are Constructors

- Functions with the name of the class
- No return type
- Maybe parameters
- Maybe more than one





How to use objects

- First step: creating the object
  - create Ball myBall ← Object's name
- Manipulating objects
  - myBall.changeColor BLUE
  - if myBall.isVisible then myBall.moveLeft
- or

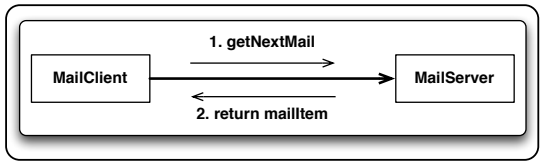
pseudocode

  - myBall.changeColor (BLUE)
  - if myBall.isVisible() then myBall.moveLeft()

Table of contents

Objects are not alone

- Use association
  - Association for exchanging messages
    - Usually one direction -> array
    - Maybe two, or unknown
  - Example: MailClients



Objects are composed

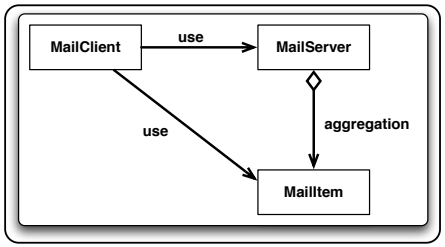
- Objects may be made of other objects
  - A car is made of an engine and 4 wheels
  - A racing team is made of a car and a driver
- This is a special case of association
  - The "is part of" association
- Two kinds of "part of" associations
  - Aggregation
  - Composition

Aggregation, the light form

- Parts exist independently from the whole objects
- Parts may belong to several aggregation at the same time
- Example:
  - A volley-ball player is part of a team
  - A young volley-ball player may be part of two teams (one in youth championship, one in adults)

Aggregation: example

- The mail system:
  - MailClients ask the MailServer for NewMails
  - The MailServer hold MailItems
  - MailClients ask MailItems for the message

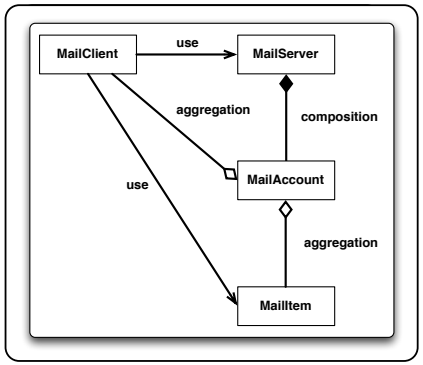


Composition, the enforced form

- An object is part of another, it cannot exist independently
- An object cannot belong to multiple objects
- Example
  - A building is composed of rooms

Composition: example

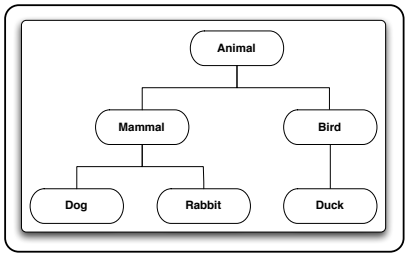
- The mail system:
  - MailAccounts are part of the mailServer
  - MailAccounts belong to MailClients
  - MailAccounts store MailItems
  - MailItems may exist out of MailAccounts



Inheritance, the essence of OOP

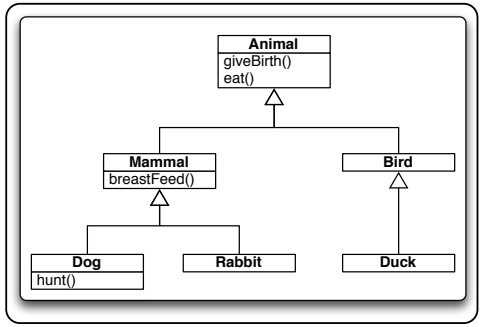
- A natural way to classify
  - A dog is a mammal, which is itself an animal
  - A rabbit too
  - A duck is a bird, which is also an animal





Inheritance, the essence of OOP

- Eliciting common features
  - Animals...
    - eat
    - give birth
  - Mammals...
    - breast feed
  - Dogs...
    - hunt



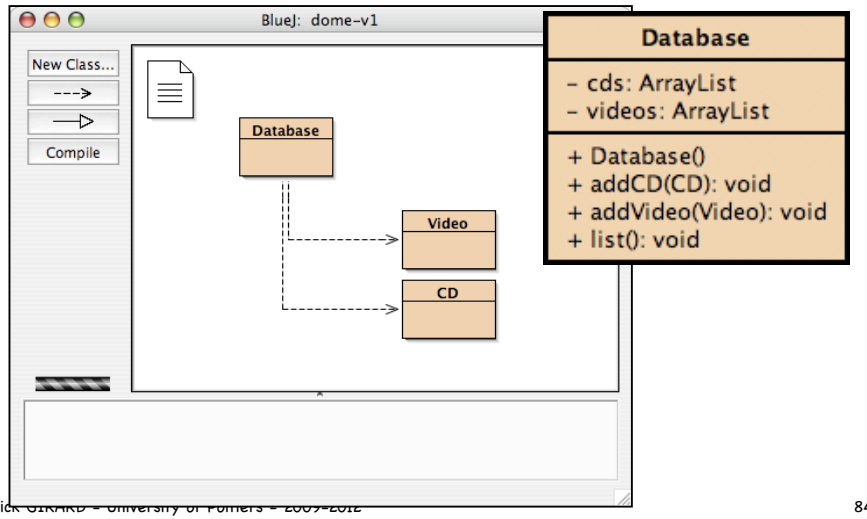
Another example

- DoME: Database of Multimedia Entertainment

Video
- title: String - director: String - playingTime: int - gotIt: boolean - comment: String
+ Video(String, String, int) + setOwn(boolean): void + getOwn(): boolean + print(): void + setComment(String): void + getComment(): String

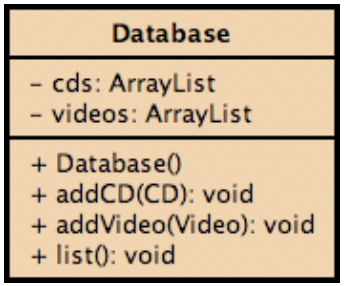
CD
- title: String - artist: String - numberOfTracks: int - playingTime: int - gotIt: boolean - comment: String
+ CD(String, String, int, int) + setOwn(boolean): void + getOwn(): boolean + print(): void + setComment(String): void + getComment(): String

Another example, DoME

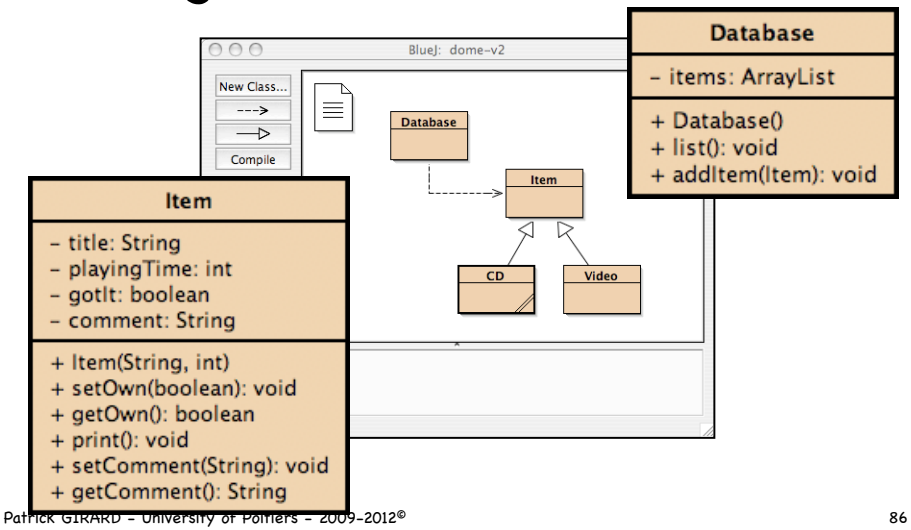


Drawbacks

- Uneasy to use
  - How can I search for a title (even in CDs or Videos) ?
- Uneasy to extend
  - How to add a "VideoGame" Category ?
  - How to introduce DVD/VHS distinction ?

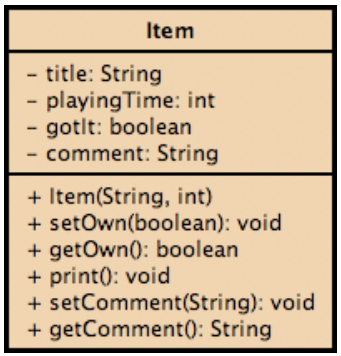


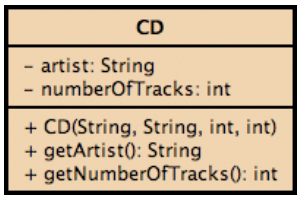
The right solution: inheritance

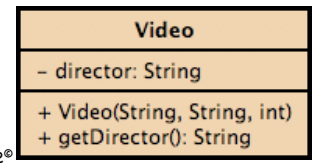


The right solution: inheritance

- The "is a" relation
  - a CD "is an" item
  - a Video "is an" item

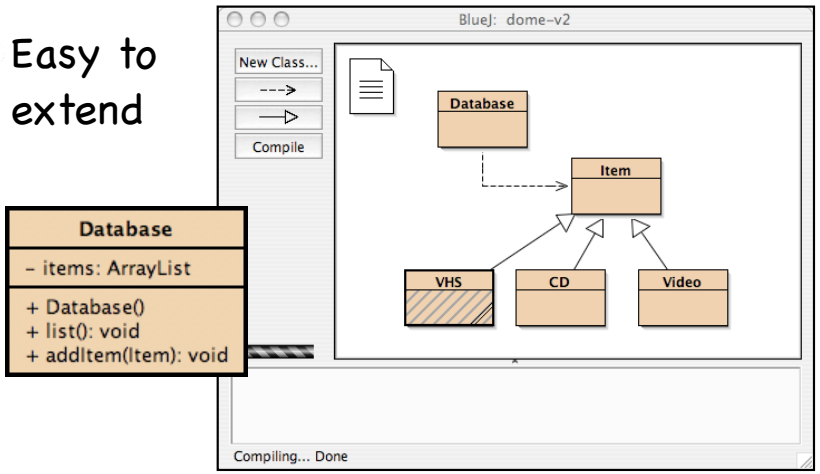






The right solution: inheritance

- Easy to extend



Inheritance, motivation

- Avoiding code duplication
- Programming by extension
- Reuse

Table of contents

Advantages

- Natural programming
- Increasing reliability
- Avoiding code duplication
- Reusing components

Table of contents

SIMULA

- 1960, Norway
- A language for doing simulations
- A framework for many object-oriented features

```
Begin
   Class Glyph;
      Virtual: Procedure print Is Procedure print;;
   Begin
   End;

   Glyph Class Char (c);
      Character c;
   Begin
      Procedure print;
        OutChar(c);
   End;
   …
```


SmallTalk

- 1970s Xerox PARC
- An interpreted language
- Full object
- First definition of widgets

```
| rectangles aPoint |
rectangles := OrderedCollection
  with: (Rectangle left: 0 right: 10 top: 100 bottom: 200)
  with: (Rectangle left: 10 right: 10 top: 110 bottom: 210).
aPoint := Point x: 20 y: 20.
collisions := rectangles
  select: [:aRect | aRect containsPoint: aPoint].
```


C++, Object Pascal, Eiffel

- 1980s
- OO Programming becomes common
  - Language multiplication
  - Full OO or extensions of existing languages
- Birth of Object Oriented Design
  - OMT, OOSE, OOD

OO Normalisation

- 1990s
- OO Programming
  - C++ ANSI Normalisation
  - Java
- OO Design
  - UML