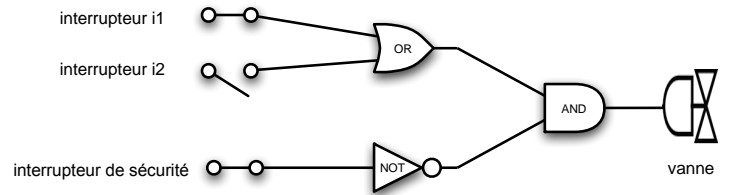


# CONCEPTS DE LA PROGRAMMATION PAR OBJETS

Objet, Classe, message, héritage (syntaxe Java)

Walter Rudametkin  
Maître de Conférences  
Bureau F011  
[Walter.Rudametkin@polytech-lille.fr](mailto:Walter.Rudametkin@polytech-lille.fr)

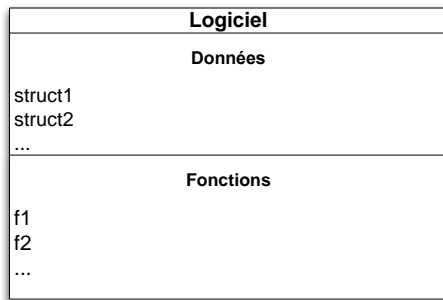
## Exemple: CAO de circuits logiques



- éditer
- afficher
- évaluer (simuler)

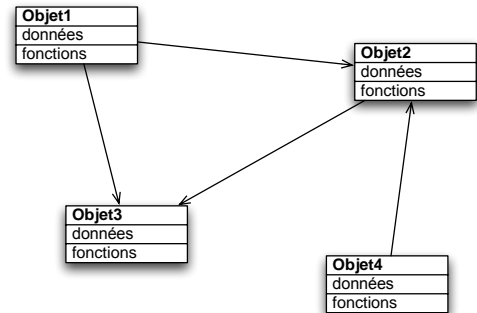
## Conception procédurale (à la C)

Dichotomie données/fonctions  
Logiciel = {données} + {fonctions}



## Conception par Objets

Chaque objet = données + fonctions  
Logiciel = {objets interagissants}



## Conception par Objets

- Données
  - **Variables d'instance** (champs, attributs, variables d'instance, "structure" interne de l'objet).
  - Détermine son état.
- Fonctions
  - **Méthodes** (procédures, fonctions)
  - Traitements que sait réaliser l'objet, « comportement de l'objet »
  - Dans son environnement local de variables

## Classe

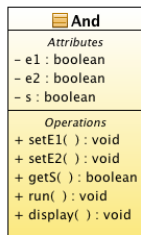
- Tout objet est « instance » d'une classe
- Classe => type d'objet

```
class <Nom de classe> {  
  
    // champs ou variables d'instance  
    <type> <var1> [, <var2> ... ] ;  
  
    // methodes  
    <type|void> <nom> ([<type> <pm1> [, <type <pm2>...]]) {  
        <declarations locales>  
        <corps>  
    }  
}
```

## Exemples

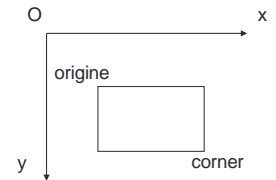
```
class And {
// variables d'instances
  boolean e1, e2, s;
  Point position;

// methodes
  void setE1(boolean etat) {
    e1 = etat;}
  void setE2(boolean etat) {
    e2 = etat;}
  boolean getS() {
    return s;}
  void run() {
    s = e1 && e2;}
  void display() ...
}
```



## Exemples

```
class Rectangle {
// champs
  Point origine, corner;
// methodes
  double largeur() {...}
  double longueur() {...}
  double surface() {...}
  void display() {...}
}
```



## Instance

- Tout objet est instance d'une classe.
- Création dynamique par instanciation :  

```
new <Classe>()
```
- Les objets instances d'une même classe disposent d'un jeu des *variables d'instance*, et des *méthodes* décrites par celle-ci.
- Ils ne diffèrent que par leur *état*, c'est à dire par les valeurs de leurs variables d'instance: *valeurs d'instance*.

## Exemple

```
{
// programmes utilisateurs ...
// variables
  And a;
  Rectangle rec;

// instanciation des classes precedentes
  a = new And();
  rec = new Rectangle();
  ...
}
```

## Message

Programme = {objets interagissants}

- Protocole d'un objet = {méthodes applicables} défini par sa classe.
- Interaction par envoi de message :  

```
<objet destinataire>.<nom méthode>(<arguments>)
```
- Mécanisme d'exécution :  
 L'objet destinataire applique la méthode <nom méthode> dans son environnement local = {ses variables d'instances}



## Exemples

```
// sachant booleens initialise a false
a.setE1(true);
a.run();
a.getS() // --> false
a.setE2(true);
a.run();
a.getS() //--> true

class Rectangle {
  Point origine, corner;
  // programmer les methodes :
  double largeur() ???
  double longueur() ???
}
```

## Mode de programmation

Orienté objet	Orienté procédure
<code>a.setE1(true)</code>	<code>setE1(a, true)</code>
<code>a.run()</code>	<code>runAnd(a)</code>
<code>a.display()</code>	<code>displayAnd(a)</code>
<code>rec.display()</code>	<code>displayRectangle(rec)</code>

### ■ Polymorphisme

- Une même opération peut prendre plusieurs formes (arithmétique, read/write...)
- PPO : c'est l'objet qui détermine l'opération.

### ■ Encapsulation

- Pas de manipulation directe de l'état de l'objet (pas d'accès aux champs) : passage par son protocole
- Modularité...

## Composition de méthodes

### • A l'intérieur de l'objet

- accès à son propre environnement (ses variables)
- et à ses propres méthodes par envoi de message à lui-même = this-message  
`this.<nom de méthode>(<arguments>)`

### • this

- dénote l'objet courant (en cours d'exécution d'une méthode)
- this n'a de sens qu'à l'intérieur d'un objet et ne peut apparaître que dans le corps de ses méthodes pour invoquer ses propres méthodes.
- simplification syntaxique (this est implicite dans tout appel de méthode à l'intérieur de l'objet) :  
`<nom de méthode>(<arguments>)`

## Exemples

```
class And {
// methodes ...
void run() {
    s = e1 && e2;
}
boolean getS() {
    this.run(); // ou plus simplement run();
    return s;
}
...
}

class Rectangle {
// methodes
double largeur() {...}
double longueur() {...}
double surface() {
    return this.largeur() * this.longueur();
}
double perimetre() {
    return 2*(largeur() + longueur());
}
...
}
```

## Sous-classe et héritage

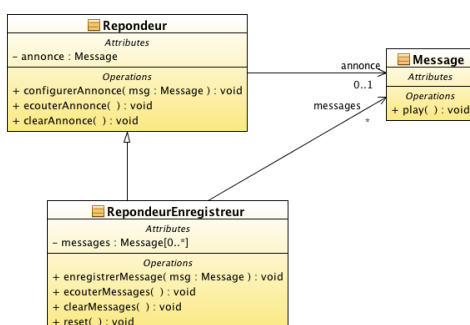
- Classe : définition d'objets semblables dans leur structure et comportement
- Sous-classe : définition d'objets partiellement semblables

```
class <Sous-classe> extends <Sur-classe> {
// variables d'instance
// methodes
}
```

### • Héritage (1)

La sous-classe dispose des caractéristiques, variables et méthodes, de la sur-classe

## Exemple



## Exemple

```
public class Repondeur {

    Message annonce;

    void configurerAnnonce(Message msg) {
        annonce = msg;
    }
    void ecouterAnnonce() {
        annonce.play();
    }
    void clearAnnonce() {
        annonce = null;
    }
}
}
```

## Exemple

```
public class RepondeurEnregistreur extends Repondeur {
    List<Message> messages;

    void enregistrerMessage(Message msg) {
        messages.add(msg);
    }
    void écouterMessages() {
        for (Message m : messages) {
            m.play();
        }
    }
    void clearMessages() {
        messages.clear();
    }
    void reset() {
        this.clearAnnonce();
        this.clearMessages();
    }
}
```

## Redéfinition de méthodes

### • Redéfinition de méthode

Il est possible de redéfinir une méthode dans une sous-classe par masquage : même nom, même profil de paramètres

### • Héritage (2)

L'héritage des méthodes est ascendant: « lookup ». Le lookup retient la première définition rencontrée = la plus spécifique.

## Exemple

```
class Repondeur {
    ...
    void send(Message msg) { // primitive interne... }
    void repondre() {
        this.send(annonce);
    }
}

class RepondeurEnregistreur extends Repondeur {
    ...
    Message getMessage() { // primitive interne... }
    void repondre() {
        this.send(annonce); // ou super.repondre()
        this.enregistrerMessage(this.getMessage());
    }
}
```

## Redéfinition incrémentale : super

:-) Redéfinition incrémentale de méthode  
=> this-message + incrément de traitement.

:-( Redéfinition = masquage  
=> la définition héritée n'est plus accessible par this-message.  
super.<nom de méthode>(<arguments>)

- «super» force l'héritage (le lookup) à rechercher la méthode à partir de la surclasse.
- Tout comme this, super dénote l'objet en cours d'exécution de la méthode et n'a de sens que dans le corps de ses propres méthodes.

### • Héritage (3)

Le lookup peut être forcé par super-message.

## Hierarchie (arbre) de classes

### • Object

Toute classe est sous-classe au minimum (et par défaut) de Object.

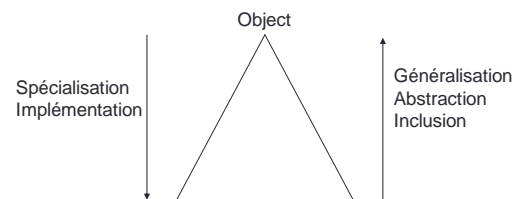
```
class Object {
    // methodes
    boolean equals (Object obj);
    String toString();
    int hashCode();
    ...
}

class Repondeur { ... }
class Repondeur extends Object { ... }
```

### • Héritage (4)

L'héritage est récursif sur la branche (chemin d'héritage) <classe>...<Object>

## Hierarchie (arbre) de classes



Classe comme...	sous-classe de
ensemble d'objets (type)	inclusion (sous-type) « est un »
code d'objets	généralisation/ spécialisation abstraction/ implémentation