Introduction à la Programmation Java : Généralités sur le polymorphisme objet

Frédéric Gava

L.A.C.L
Laboratoire d'**A**lgorithmique, **C**omplexité et **L**ogique

Cours de L3 MIAGE

Plan

Plan

Plan

Déroulement du cours

Le problème des suites

Suite de Fibonacci

Une suite de Fibonacci est un objet contenant :

1. un état : la valeur de 2 variables a et b ;
2. un fonctionnement : on peut demander à l'objet l'élément suivant de la suite, c'est-à-dire que l'on reçoit alors la nouvelle valeur de la suite et l'on calcul l'élément successeur.

Code Java

```
public class Fibonacci
 {
  private int a=0, b=1;
  public int EltSuivant()
   {
    int tmp=a;
    a=a+b;
    b=tmp;
    return b;
   }
 }
```

La classe Fibonacci représente un type : le type Fibonacci. Nous pouvons déclarer des références de ce type.

Le problème des suites

Suite de Fibonacci

Une suite de Fibonacci est un objet contenant :

1. un état : la valeur de 2 variables a et b ;
2. un fonctionnement : on peut demander à l'objet l'élément suivant de la suite, c'est-à-dire que l'on reçoit alors la nouvelle valeur de la suite et l'on calcul l'élément successeur.

Code Java

```
public class Fibonacci
 {
  private int a=0, b=1;
  public int EltSuivant()
   {
    int tmp=a;
    a=a+b;
    b=tmp;
    return b;
   }
 }
```

La classe Fibonacci représente un type : le type Fibonacci. Nous pouvons déclarer des références de ce type.

Le problème des suites

Utilisation des suites de Fibonacci

```
//Déclaration
Fibonacci fibo1, fibo2, fibo3;

//Instanciation
fibo1 = new Fibonacci();
fibo2 = new Fibonacci();
fibo3 = new Fibonacci();

//Affichage
System.out.println(fibo1.EltSuivant());
System.out.println(fibo2.EltSuivant());
System.out.println(fibo1.EltSuivant());
```

Autres suites

Il est facile d'introduire d'autres classes de suites sur le style de la suite de Fibonacci :

```
public class Lineaire
{
 private int pente=0, i=0;
 public Lineaire(int pente) { this.pente=pente; }
 public int EltSuivant() { return pente*i++; }
 public int pente() { return pente; }
}
```

Le problème des suites

Utilisation des suites de Fibonacci

```
//Déclaration
Fibonacci fibo1, fibo2, fibo3;

//Instanciation
fibo1 = new Fibonacci();
fibo2 = new Fibonacci();
fibo3 = new Fibonacci();

//Affichage
System.out.println(fibo1.EltSuivant());
System.out.println(fibo2.EltSuivant());
System.out.println(fibo1.EltSuivant());
```

Autres suites

Il est facile d'introduire d'autres classes de suites sur le style de la suite de Fibonacci :

```
public class Lineaire
{
 private int pente=0, i=0;
 public Lineaire(int pente) { this.pente=pente; }
 public int EltSuivant() { return pente*i++; }
 public int pente() { return pente; }
}
```

Le problème des suites

Suite Affine

Nous disons qu'une suite est affine si :

- comme toute suite lineaire, on peut lui demander l'élément suivant
- comme une suite linéaire, elle possède une pente
- et en outre, elle possède une cote qui la déporte par rapport à une suite linéaire ordinaire

Implantation

```
public class Affine
{
 private int pente, cote, i=0;
 public Affine(int pente, cote)
  {
   this.pente=pente;
   this.cote=cote;
  }

 public int EltSuivant() { return cote+pente*i++; }
}
```

Le problème des suites

Suite Affine

Nous disons qu'une suite est affine si :

- comme toute suite lineaire, on peut lui demander l'élément suivant
- comme une suite linéaire, elle possède une pente
- et en outre, elle possède une cote qui la déporte par rapport à une suite linéaire ordinaire

Implantation

```
public class Affine
{
 private int pente, cote, i=0;
 public Affine(int pente, cote)
  {
   this.pente=pente;
   this.cote=cote;
  }

 public int EltSuivant() { return cote+pente*i++; }
}
```

Le problème des suites

Héritage

Cependant, nous avons pris soin de définir cette classe comme la classe linéaire mais avec une propriété en plus :

nous pouvons la dériver de cette dernière par **héritage**.

Nouvelle Implantation

```
public class Affine extends Lineaire
{
 private int cote;
 //pas de pente car provient de la classe mère Lineaire

 public Affine(int pente, cote)
  {
   super(pente); // utilisation du constructeur de la classe mère
   this.cote=cote;
  }

 public int EltSuivant() { return cote+pente*i++; }
 //pas de méthode pente car provient de la classe mère Lineaire
}
```

Le problème des suites

Héritage

Cependant, nous avons pris soin de définir cette classe comme la classe linéaire mais avec une propriété en plus :

nous pouvons la dériver de cette dernière par **héritage**.

Nouvelle Implantation

```
public class Affine extends Lineaire
{
 private int cote;
 //pas de pente car provient de la classe mère Lineaire

 public Affine(int pente, cote)
  {
   super(pente); // utilisation du constructeur de la classe mère
   this.cote=cote;
  }

 public int EltSuivant() { return cote+pente*i++; }
 //pas de méthode pente car provient de la classe mère Lineaire
}
```

Le problème des suites

Conséquence

La conséquence est double :

1. nous nous économisons un peu d'écriture de code
2. toute suite affine est "une sorte de" linéaire (on parle de sous-typage ou de polymorphisme) : le type Affine est compatible avec le type Lineaire. Partout où nous avions du Linéaire, nous pouvons utiliser de l'Affine...car Affine répondra au minimum à tout les message de Linéaire (aura au minimum les mêmes méthodes et constructeurs).

Moralité : **"qui peut le plus, peut le moins"**

Compatibilité

La compatibilité nous autorise à écrire Lineaire suite = **new** Affine (3,2);.
Ceci montre qu'à une référence comme "suite" correspond deux types :

1. le type déclaré de la référence, dit **statique**, car connu du compilateur Java ;
2. le type de l'objet référencé, dit **dynamique**, car ce type ne peut être prédit à la compilation, il ne sera connu qu'à l'exécution.

A l'exécution, l'instruction v=suite.EltSuivant(); invoque EltSuivant() sur l'objet référencé (on envoie le message

EltSuivant à l'objet) : l'exemple précèdent montre que c'est le type dynamique de l'objet qui détermine quel code

sera exécuté

Le problème des suites

Conséquence

La conséquence est double :

1. nous nous économisons un peu d'écriture de code
2. toute suite affine est "une sorte de" linéaire (on parle de sous-typage ou de polymorphisme) : le type Affine est compatible avec le type Lineaire. Partout où nous avions du Linéaire, nous pouvons utiliser de l'Affine...car Affine répondra au minimum à tout les message de Linéaire (aura au minimum les mêmes méthodes et constructeurs).

Moralité : **"qui peut le plus, peut le moins"**

Compatibilité

La compatibilité nous autorise à écrire Lineaire suite = **new** Affine (3,2);.
Ceci montre qu'à une référence comme "suite" correspond deux types :

1. le type déclaré de la référence, dit **statique**, car connu du compilateur Java ;
2. le type de l'objet référencé, dit **dynamique**, car ce type ne peut être prédit à la compilation, il ne sera connu qu'à l'exécution.

A l'exécution, l'instruction v=suite.EltSuivant(); invoque EltSuivant() sur l'objet référencé (on envoie le message

EltSuivant à l'objet) : l'exemple précèdent montre que c'est le type dynamique de l'objet qui détermine quel code

sera exécuté

Héritage

Une classe spéciale

Toute classe non déclaré comme héritant explicitement d'une autre classe, hérite implicitement d'une classe particulière du langage Java appellée Object :

```
public class Object
 {
  ...
  public String toString(){ ... }
  ...
 }
```

Remarquons qu'Object hérite alors lui même d'Object, ce qui n'est pas génant en Java...

Conséquences

Donc :

- toute les méthodes d'Object (comme toString) peuvent être invoquée sur tout objet (envoie de message générique)
- toute classe peut (et devrait) redéfinir utilement ces méthodes (comme toString)

Remarquons que String est une classe, donc hérite d'Object donc on peut avoir String.toString. Rigolo non ?

Héritage

Une classe spéciale

Toute classe non déclaré comme héritant explicitement d'une autre classe, hérite implicitement d'une classe particulière du langage Java appellée Object :

```
public class Object
 {
  ...
  public String toString(){ ... }
  ...
 }
```

Remarquons qu'Object hérite alors lui même d'Object, ce qui n'est pas génant en Java...

Conséquences

Donc :

- toute les méthodes d'Object (comme toString) peuvent être invoquée sur tout objet (envoie de message générique)
- toute classe peut (et devrait) redéfinir utilement ces méthodes (comme toString)

Remarquons que String est une classe, donc hérite d'Object donc on peut avoir String.toString. Rigolo non ?

Le problème des suites

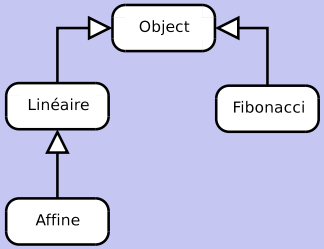
Nouveau Code

```
public class Lineaire
 {
  ...
  public String toString()
   {
    return getClass().getName() + ", pente="+pente;
   }
}

public class Affine extends Lineaire
{
 ...
 public String toString()
  {
   return "Affine, pente=" + pente + ", cote=" + cote
  }

 // autre possibilité

 public String toString()
  {
   return super.toString()+ "cote=" + cote
  }
}
```

Interface

Hiérarchie

Nous avons actuellement la hiérarchie de classes suivante :



Or les classes Lineaire et Fibonacci ont la même méthode EltSuivant : elle peut donc répondre au même message.

Ceci est lié au fait que nous pouvons, dans notre contexte, poser ma définition : une suite est un objet dont on peut demander l'élément suivant.

Première interface

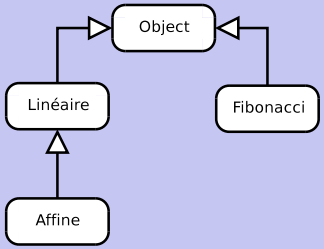
Nos traduirons cet énoncé en une sorte de classe sans aucun code exécutable : une interface.

```
public interface Suite
{
 public int EltSuivant();
}
```

Interface

Hiérarchie

Nous avons actuellement la hiérarchie de classes suivante :



Or les classes Lineaire et Fibonacci ont la même méthode EltSuivant : elle peut donc répondre au même message.

Ceci est lié au fait que nous pouvons, dans notre contexte, poser ma définition : une suite est un objet dont on peut demander l'élément suivant.

Première interface

Nos traduirons cet énoncé en une sorte de classe sans aucun code exécutable : une interface.

```
public interface Suite
{
 public int EltSuivant();
}
```

Interface

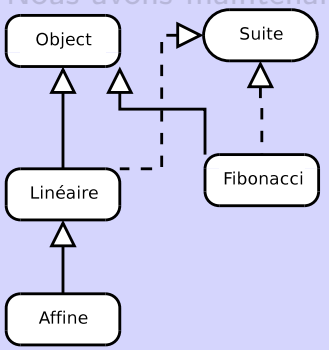
Réalisation

Les classes Lineaire et Fibonacci sont deux réalisation (2 implantations) de Suite, ce que nous pouvons traduire en Java :

```
public class Lineaire implements Suite {
 public int EltSuivant() { ...}
 ...
}

public class Fibonacci implemens Suite {
 public int EltSuivant() { ...}
 ...
}
```

Nouvelle hiérarchie

Nous avons maintenant une hiérarchie de classes et d'interfaces comme ceci :



Interface

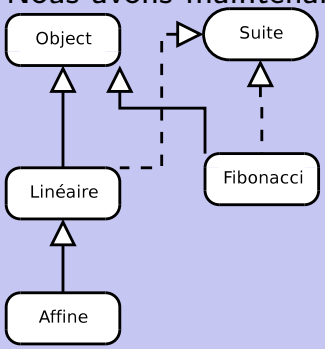
Réalisation

Les classes Lineaire et Fibonacci sont deux réalisation (2 implantations) de Suite, ce que nous pouvons traduire en Java :

```
public class Lineaire implements Suite {
 public int EltSuivant() { ...}
 ...
}

public class Fibonacci implemens Suite {
 public int EltSuivant() { ...}
 ...
}
```

Nouvelle hiérarchie

Nous avons maintenant une hiérarchie de classes et d'interfaces comme ceci :



Interface

Utilisation

Nous pouvons maintenant écrire une classe générique (polymorphisme objet) Formateur qui affiche les *n* premiers éléments d'une suite :

```
public class Formateur
{
 public String formater(Suite suite, int n)
  {
   String texte = "Suite " + suite.toString() + " : ";
   for (int i=0; i<n; i++)
    texte= texte + " " + suite.EltSuivant();
   return texte;
  }
}
```

Utilisation

La liaison dynamique utilise les types statiques dynamiques (on rappel qu'en Java tout est fortement typé)

```
Suite[] table = new Suite[3]; // 3 références (type statiques) Suite
table[0] = new Lineaire(3); // objet (type dynamique) Lineaire
table[1] = new Affine(3,7); // objet (type dynamique) Affine
table[2] = new Fibonacci(); // objet (type dynamique) Fibonacci
```

C'est toujour le type dynamique qui détermine le code qui sera réellement executé

Le transtypage

Transtyper un objet

Dans le cas Object objet = **new** Affine(−3,100); l'objet référencé par objet est de type dynamique Affine. Il est donc possible d'invoquer sur cet objet les méthode d'Affine mais non sur la référence objet qui est de type Object : le compilateur refusera par exemple l'invocation objet.cote().

Pour traiter une référence d'un certain type statique comme si elle était d'un autre type, il faut transtyper par un

"cast" ((Affine)objet).cote();.

Le problème

Cette sorte de transtypage est toujour vérifiée à l'exécution : il faut que le type de l'objet (type dynamique) hérite (on parle de sous-typage) du type résultant du transtypage. Dans le cas présent les transtypage suivants sont correctes :

```
((Objet)objet).toString();
((Suite)objet).toString; //ou .EltSuivant()
((Lineaire)objet).toString(); //ou .EltSuivant() ou .pente()
((Affine)objet).toString(); //ou .EltSuivant() ou .pente() ou .cote()
```

Mais ceci n'est pas valide :

```
Lineaire objet = new Fibonacci(); // erreur à la compilation

Object objet = new Affine(−3,100);
((Fibonacci)objet).EltSuivant(); // erreur à la compilation

Lineaire objet = new Lineaire();
((Fibonacci)objet).EltSuivant(); // erreur à la compilation
```

Le transtypage

Transtyper un objet

Dans le cas Object objet = **new** Affine(−3,100); l'objet référencé par objet est de type dynamique Affine. Il est donc possible d'invoquer sur cet objet les méthode d'Affine mais non sur la référence objet qui est de type Object : le compilateur refusera par exemple l'invocation objet.cote().

Pour traiter une référence d'un certain type statique comme si elle était d'un autre type, il faut transtyper par un

"cast" ((Affine)objet).cote();.

Le problème

Cette sorte de transtypage est toujour vérifiée à l'exécution : il faut que le type de l'objet (type dynamique) hérite (on parle de sous-typage) du type résultant du transtypage. Dans le cas présent les transtypage suivants sont correctes :
((Objet)objet).toString();
((Suite)objet).toString; *//ou .EltSuivant()*
((Lineaire)objet).toString(); *//ou .EltSuivant() ou .pente()*
((Affine)objet).toString(); *//ou .EltSuivant() ou .pente() ou .cote()*

Mais ceci n'est pas valide :
Lineaire objet = **new** Fibonacci(); *// erreur à la compilation*

Object objet = **new** Affine(−3,100);
((Fibonacci)objet).EltSuivant(); *// erreur à la compilation*

Lineaire objet = **new** Lineaire();
((Fibonacci)objet).EltSuivant(); *// erreur à la compilation*

Retour sur les exceptions

Exemple

```
InputStream is = new FileInputStream("source.txt");
try
 {
  while ((car=is.read()) != -1)
   {
    ...
   }
 } catch (IOException e)
     {
      System.err.println(e);
     }
```

Une méthode susceptible de voir "lancer" dans son code une exception a deux solution :

1. prévoir un bloc d'interception similaire à l'exemple
2. ne pas intercepter et signaler qu'elle risque de lancer elle-même une exception lorsqu'on l'invoque (remarquez que 1) et 2) ne sont pas imcompatible, on peut écrire un bloc pour relancer l'exception)

Solution

Dans ce dernier cas, la méthode doit impérativement le signalier dans sa signature :

```
public void lireFichier(String source) throws IOException { ... }
```

sauf si l'exception fait partie des RuntimeException (comme IndexOutOfBoundsException).

Retour sur les exceptions

Exemple

```
InputStream is = new FileInputStream("source.txt");
try
 {
  while ((car=is.read()) != -1)
   {
    ...
   }
 } catch (IOException e)
     {
      System.err.println(e);
     }
```

Une méthode susceptible de voir "lancer" dans son code une exception a deux solution :

1. prévoir un bloc d'interception similaire à l'exemple
2. ne pas intercepter et signaler qu'elle risque de lancer elle-même une exception lorsqu'on l'invoque (remarquez que 1) et 2) ne sont pas imcompatible, on peut écrire un bloc pour relancer l'exception)

Solution

Dans ce dernier cas, la méthode doit impérativement le signalier dans sa signature :

```
public void lireFichier(String source) throws IOException { ... }
```

sauf si l'exception fait partie des RuntimeException (comme IndexOutOfBoundsException).

Déroulement du cours

Programme principal

Classe de lancement

N'importe quelle classe peut être utiliser pour être le point d'entrée du programme (on peut même en mettre plusieurs, une par classe). Pour cela, il suffit de rajouter à la classe, la méthode statique main(String[] argv) :

```
public class Suites {
 static public void main(String[] argv)
   {
     System.out.println("coucou");
   }
}
```

Applet

Class de lancement internet

On peut avoir un programme Java exécuter par les butinneurs :

```
import java.awt.*
import java.applet.*
public class SalutWeb extends Applet
 {
  public void paint(Graphics g)
   {
    g.drawString("Salut Web", 10, 10);
   }
 }
```

Code HTML

```
<html>
<head>
 <title> Un essai </tiltle>
</head>
<body>
 <applet code="SalutWeb" width=300 height=50>  </applet>
</body>
```

Applet

Class de lancement internet

On peut avoir un programme Java exécuter par les butinneurs :

```
import java.awt.*
import java.applet.*
public class SalutWeb extends Applet
 {
  public void paint(Graphics g)
   {
    g.drawString("Salut Web", 10, 10);
   }
 }
```

Code HTML

```
<html>
<head>
 <title> Un essai </tiltle>
</head>
<body>
 <applet code="SalutWeb" width=300 height=50>  </applet>
</body>
```

Déroulement du cours

Les carastéristiques

Classes membres, internes et anonymes

- Une classe membre est membre d'une autre classe (public, private, ...)
- une classe interne est définie localement à un bloc (visibilité locale)
- une classe anonyme n'a pas de nom et ne peut servir que dans des instantiations (opérande pour **new**)

Exemple avec classe membre

```
class Donnee
 {
  public String toString() { return "Donnee";}
 }

public class Utilisatrice
{
 class ClasseMembre extends Donne
  {
   public String toString() { return "Donnee locale";}
  }

 Donne donnee() { return new ClasseMembre();}
}
```

Les carastéristiques

Classes membres, internes et anonymes

- Une classe membre est membre d'une autre classe (public, private, ...)
- une classe interne est définie localement à un bloc (visibilité locale)
- une classe anonyme n'a pas de nom et ne peut servir que dans des instantiations (opérande pour **new**)

Exemple avec classe membre

```
class Donnee
 {
  public String toString() { return "Donnee";}
 }

public class Utilisatrice
{
 class ClasseMembre extends Donne
  {
   public String toString() { return "Donnee locale";}
  }

 Donne donnee() { return new ClasseMembre();}
}
```

Les carastéristiques

Exemple avec classe locale à un bloc

```
public class Utilisatrice
{
 Donnee donnee()
  {
   class ClasseInterne extends Donnee { public String toString(){ return "Donnee interne";} };
   return new ClasseInterne();
  }
}
```

Exemple avec classe anonyme directement instanciée

```
public class Externe
{
 Donnee donnee() { return new Donnes() { public String toString() { return "Donnee anonyme";} }; }
}
```

Dans ce cas, il faut comprendre que ... **new** TelleClasseOuInterface() crée un objet instancie d'une classe héritière de TelleClasseOuInterface, celle classe déclare ou redéclare les membres explicités et garde les autres tels quels.

On obtient ainsi des "objets courant emboîtés" ; le langage Java permet de préfixer **new**, **this** et **super** par le nom

de la classe concerné, si ce n'est pas la plus interne. Exemple : Englobante.**this**.telleMethode();

Gestion des noms (ou contrôle d'accès)

Différents paquetages

Les classe de la bibliothèque standard (API Java) sont regroupées dans divers répertoires appelés paquetages (ou encore archive Java=.jar), réunis notamment dans les répertoires java et javax. Exemples :

java.lang	⇒	classes fondamentales
java.io	⇒	clases gérant les entrées-sorties
java.awt	⇒	classes d'interfaces fenêtrées
java.awt.event	⇒	classes d'événement utilisés dans awt
java.applet	⇒	classe des applets
javax.swing	⇒	classes de constructions d'IHM (interfaces fenêtrées améliorées)
org.omg.CORBA	⇒	portage Java de l'API CORBA (calculs distribués)
etc.		

Utilisation

Ces répertoires java sont connus du navigateur et du compilateur java par la variable d'environnement CLASSPATH (moins utile depuis JDK 1.2). Lorsqu'un fichier source a besoin d'utiilser le service d'une classe appartenant au paquetage p, elle peut le faire de 2 façons :

1. préfixer le nom par le chemin d'accès complet (ex. java.awt.event.MouseEvent)
2. "importer" le paquetage avant toute déclaration de classe : **import** java.awt.event.MouseEvent ou **import** java.awt.event.∗ //∗ == *"toutes les classes du paquetage"*

Exemple : la classe SalutWeb avait besoin d'accéder aux classes Graphics (dans java.awt) et à Applet (dans java.applet). Sans les clauses d'importations nous aurions du écrire :

Gestion des noms (ou contrôle d'accès)

Différents paquetages

Les classe de la bibliothèque standard (API Java) sont regroupées dans divers répertoires appelés paquetages (ou encore archive Java=.jar), réunis notamment dans les répertoires java et javax. Exemples :

java.lang	⇒	classes fondamentales
java.io	⇒	clases gérant les entrées-sorties
java.awt	⇒	classes d'interfaces fenêtrées
java.awt.event	⇒	classes d'événement utilisés dans awt
java.applet	⇒	classe des applets
javax.swing	⇒	classes de constructions d'IHM (interfaces fenêtrées améliorées)
org.omg.CORBA	⇒	portage Java de l'API CORBA (calculs distribués)
etc.		

Utilisation

Ces répertoires java sont connus du navigateur et du compilateur java par la variable d'environnement CLASSPATH (moins utile depuis JDK 1.2). Lorsqu'un fichier source a besoin d'utiilser le service d'une classe appartenant au paquetage p, elle peut le faire de 2 façons :

1. préfixer le nom par le chemin d'accès complet (ex. java.awt.event.MouseEvent)
2. "importer" le paquetage avant toute déclaration de classe : **import** java.awt.event.MouseEvent ou **import** java.awt.event.∗ //∗ == *"toutes les classes du paquetage"*

Exemple : la classe SalutWeb avait besoin d'accéder aux classes Graphics (dans java.awt) et à Applet (dans java.applet). Sans les clauses d'importations nous aurions du écrire :

Unités compilables

Déclaration

Toute classe est déclaré dans un fichier, un fichier peut déclarer plusieurs classes. Un tel fichier est une unit" compilable. Une unité compilable peut se déclarer membre d'un paquetage par la déclaration : **package** p; tout au début du fichier source.

Le (les) fichier .claa correspondant doit alors obligatoirement être placé dans le répertoire p pour que le paquetage

et le nom de la classe corresponde au chemin d'accès réel du fichier compilé.

Paquetage et classe publique

Une classe peut être déclarée publique ou non publique (**public class** Point ...).
Une unité compilable java ne peut contenir qu'une seule classe publique car Jave exige que le fichier et celle classe portent le même nom. Seule une classe publique peut être utilisée en dehors du paquetage où elle appartient. Il y a deux raisons d'offrir des paquetages :

1. organisation, compréhension de l'ensemble des milliers de classes
2. encapsulation, abstraction par la division en classes publiques et non publiques

S

Unités compilables

Déclaration

Toute classe est déclaré dans un fichier, un fichier peut déclarer plusieurs classes. Un tel fichier est une unit" compilable. Une unité compilable peut se déclarer membre d'un paquetage par la déclaration : **package** p; tout au début du fichier source.

Le (les) fichier .claa correspondant doit alors obligatoirement être placé dans le répertoire p pour que le paquetage

et le nom de la classe corresponde au chemin d'accès réel du fichier compilé.

Paquetage et classe publique

Une classe peut être déclarée publique ou non publique (**public class** Point ...).
Une unité compilable java ne peut contenir qu'une seule classe publique car Jave exige que le fichier et celle classe portent le même nom. Seule une classe publique peut être utilisée en dehors du paquetage où elle appartient. Il y a deux raisons d'offrir des paquetages :

1. organisation, compréhension de l'ensemble des milliers de classes
2. encapsulation, abstraction par la division en classes publiques et non publiques

S

Membres statique, non statique

Déclaration

Un membre doit être déclaré statique ou non. Par defaut il est non statique. Un membre statique :

- n'a pas besoin de paramètre de référence
- ne peut donc pas faire usage dans son code, même implicitement de la référence **this**
- dépend seulement "de la classe", il peut être désigné ou invoqué en préfixant par le nom de la classe

Exemple

Ainsi :

```
class A
 {
  static int n=0;
  int i;
  static public void f() { i=n; //erreur à la compilation }
 }
```

L'erreur provient du fait que f est statique et ne peut donc pas fournir de référence this au champ i qui en a besoin (même implicitement).

Une méthode publique statique "main" sert de point d'entrée au programme (remarque : toute classe peut en être munie donc toute classe peut être un programme en soit).

Membres statique, non statique

Déclaration

Un membre doit être déclaré statique ou non. Par defaut il est non statique. Un membre statique :

- n'a pas besoin de paramètre de référence
- ne peut donc pas faire usage dans son code, même implicitement de la référence **this**
- dépend seulement "de la classe", il peut être désigné ou invoqué en préfixant par le nom de la classe

Exemple

Ainsi :

```
class A
 {
  static int n=0;
  int i;
  static public void f() { i=n; //erreur à la compilation }
 }
```

L'erreur provient du fait que f est statique et ne peut donc pas fournir de référence this au champ i qui en a besoin (même implicitement).

Une méthode publique statique "main" sert de point d'entrée au programme (remarque : toute classe peut en être munie donc toute classe peut être un programme en soit).

Membres public, non public

Déclaration

Un membre doit être déclaré public ou non public. Par défaut un membre est non public. Exemple :

```
public class Compteur
{
 int n=0;
 public int valeur() { return n; }
}
```

Membre publique	$\Rightarrow$	destiné à être utilisé dans des algorithmes faisant partie d'autres classes
Membre non public	$\Rightarrow$	destiné à un usage interne à la classe

Exemple

```
public class B
 {
  public vois verifier(Compteur a)
   {
    if (a.valeur()>3) { System.out.println("ok");
    if (a.n>3) { System.out.println("_pas_ok"); // erreur à la compilation
   }
 }
```

La classe B peut invoquer sur l'objet a la méthode publique (assesseur) valeur mais ne peut pas utiliser le champs a.n car n est non publique dans la classe Compteur (c'est donc une protection en écriture sur les champs d'un objet).

Membres public, non public

Déclaration

Un membre doit être déclaré public ou non public. Par défaut un membre est non public. Exemple :

```
public class Compteur
{
 int n=0;
 public int valeur() { return n; }
}
```

Membre publique	$\Rightarrow$	destiné à être utilisé dans des algorithmes faisant partie d'autres classes
Membre non public	$\Rightarrow$	destiné à un usage interne à la classe

Exemple

```
public class B
 {
  public vois verifier(Compteur a)
   {
    if (a.valeur()>3) { System.out.println("ok");
    if (a.n>3) { System.out.println("␣pas␣ok"); // erreur à la compilation
   }
 }
```

La classe B peut invoquer sur l'objet a la méthode publique (assesseur) valeur mais ne peut pas utiliser le champs a.n car n est non publique dans la classe Compteur (c'est donc une protection en écriture sur les champs d'un objet).

Membres public, non public

Et si pas public ?

Il y a 3 sortes de membres non publics :

1. sans mention ⇒ accessible du paquetage
2. **private** ⇒ accessible que de la classe
3. **protected** ⇒ accessible du paquetage et des classes héritières

Exemple

```
class NomCourt extends Exception
 {
  public NomCourt(String s)
   {
    super(s);
   }

  public String toString()
   {
    return "Le nom "+ getMessage() + "est très court";
   }
 }
```

Membres public, non public

Et si pas public ?

Il y a 3 sortes de membres non publics :

1. sans mention ⇒ accessible du paquetage
2. **private** ⇒ accessible que de la classe
3. **protected** ⇒ accessible du paquetage et des classes héritières

Exemple

```
class NomCourt extends Exception
 {
  public NomCourt(String s)
   {
    super(s);
   }

  public String toString()
   {
    return "Le nom "+ getMessage() + "est très court";
   }
 }
```

A la semaine prochaine