INTI
FORMATION

Support de cours
INTRODUCTION à la POO

INSTITUT DES NOUVELLES TECHNOLOGIES DE L'INFORMATION

LA PROGRAMMATION ORIENTEE OBJET,UN NOUVEAU CONCEPT DE DEVELOPPEMENT

"Au cours des 35 dernières années, les concepteurs de matériel informatique sont passés de machines de la taille d'un hangar à des ordinateurs portables légers basés sur de minuscules microprocesseurs. Au cours des mêmes années, les développeurs de logiciels
sont passés de l'écriture de programmes en assembleur et en COBOL à
l'écriture de programmes encore plus grands en C et C++. On pourra parler de progrès (bien que cela soit discutable), mais il est clair que le monde du logiciel ne progresse pas aussi vite que celui du matériel. Qu'ont donc les développeurs de matériel que les
développeurs de logiciels n'ont pas ?
La réponse est donnée par les composants. Si les ingénieurs en matériel électronique devaient partir d'un tas de sable à chaque fois qu'ils conçoivent un nouveau dispositif, si leur première étape devait toujours consister à extraire le silicium pour fabriquer des circuits intégrés, ils ne
progresseraient pas bien vite.
Or, un concepteur de matériel construit toujours un système à partir de composants préparés, chacun chargé d'une fonction particulière et fournissant un ensemble de services à travers des interfaces définies. La tâche des concepteurs de matériel est considérablement simplifiée par le travail de leurs prédécesseurs.
La réutilisation est aussi une voie vers la création de meilleurs logiciels.
Aujourd'hui encore, les développeurs de logiciels en sont toujours à partir d'une certaine forme de sable et à suivre les mêmes étapes que les centaines de programmeurs qui les ont précédés. Le résultat est souvent excellent, mais il pourrait être amélioré. La création de nouvelles applications à partir de composants existants, déjà testés, a toutes chances de produire un code plus fiable. De plus, elle peut se révéler nettement plus rapide et plus économique, ce qui n'est pas moins important."
La notion d'objets date du projet du missile MINUTEMAN en 1957.
La conception et la simulation du fonctionnement de ce missile reposait sur une poignée de composants logiciels prenant en charge la partie physique du missile, les trajectoires, les différentes parties du vol, ...
Le fonctionnement du système global reposait sur l'échange de messages d'information entre les différents composants.
Chaque composant logiciel était conçu par un spécialiste et possédait ses données privées. De même, le composant était virtuellement isolé du reste du programme par l'ensemble de ses méthodes servant d'interface.

LES METHODES OBJET

La modélisation objet consiste à créer une représentation informatique des éléments du monde réel auxquels on s'intéresse, sans se préoccuper de l'implémentation, ce qui signifie indépendamment d'un langage de programmation. Il s'agit donc de déterminer les objets présents et d'isoler leurs données et les fonctions qui les utilisent. Pour cela des méthodes ont été mises au point. Entre 1970 et 1990, de nombreux analystes ont mis au point des approches orientées objets, si bien qu'en 1994 il existait plus de 50 méthodes objet.
Toutefois seules 3 méthodes ont véritablement émergé:
La méthode **OMT** de *Rumbaugh*
La méthode **BOOCH'93** de *Booch*
La méthode **OOSE** de *Jacobson*

A partir de 1994, Rumbaugh et Booch (rejoints en 1995 par Jacobson) ont uni leurs efforts pour mettre au point le langage de description UML (Unified Modeling Language), qui permet de définir un langage standard en incorporant les avantages des différentes méthodes précédentes (ainsi que celles d'autres analystes). Il permet notamment de "programmer" entièrement une application avec un langage qui modélise toutes les composantes du futur programme.

LE PARADIGME OBJET ET LES 3 PRINCIPES FONDAMENTAUX DE LA PROGRAMMATION ORIENTEE OBJET

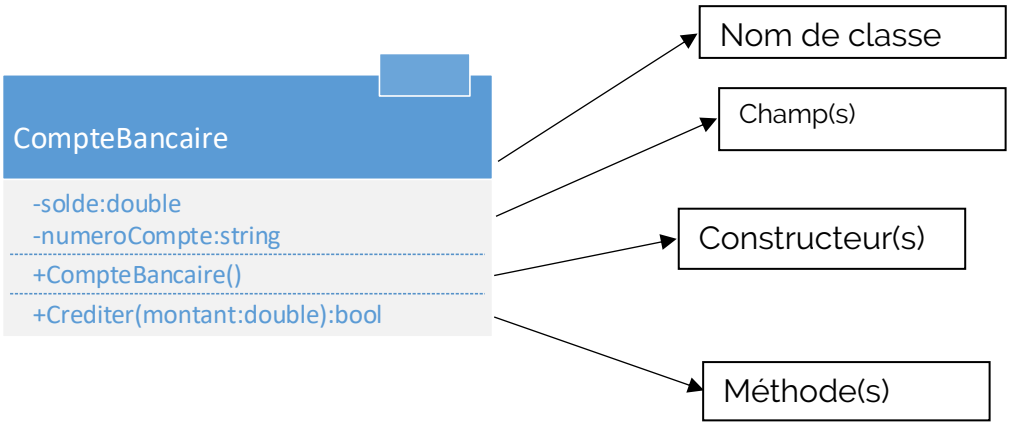
La POO est en fait une évolution de la programmation modulaire, elle apporte principalement trois aspects.

- **L'encapsulation** : Cette méthode permet de réunir des variables et des fonctions au sein d'une même entité nommée classes. Les variables sont appelées « champs » et les fonctions «méthodes », L'accès aux données et méthodes peut être aussi réglementé (grâce aux mots clés public private et protected).
- ***L'héritage*** : cette technique permet de définir une hiérarchie de classe(ou graphe). Chaque classe fille hérite des méthodes et des données de sa "mère". En pratique, la classe de base est une classe générique et souvent abstraite, ainsi plus on descend dans la hiérarchie, plus on spécialise cette classe.
- ***Le polymorphisme*** : ce nom qui vient du grec signifie "peut prendre plusieurs formes". Cette caractéristique offre la possibilité de définir plusieurs méthodes de même nom mais possédant des paramètres différents. La bonne méthode est choisie en fonction de ses paramètres lors de l'appel. Par ailleurs, il existe plusieurs types de polymorphisme :

    - Le polymorphisme ad hoc (également *surcharge* (*overloading*))
    - Le polymorphisme de données(Universel)
        - Le polymorphisme paramétrique (également *généricité* (anglais *template)*)
        - Le polymorphisme d'héritage (également *redéfinition*, *spécialisation* (*overriding*))

PRESENTATION DE LA NOTION D'OBJET

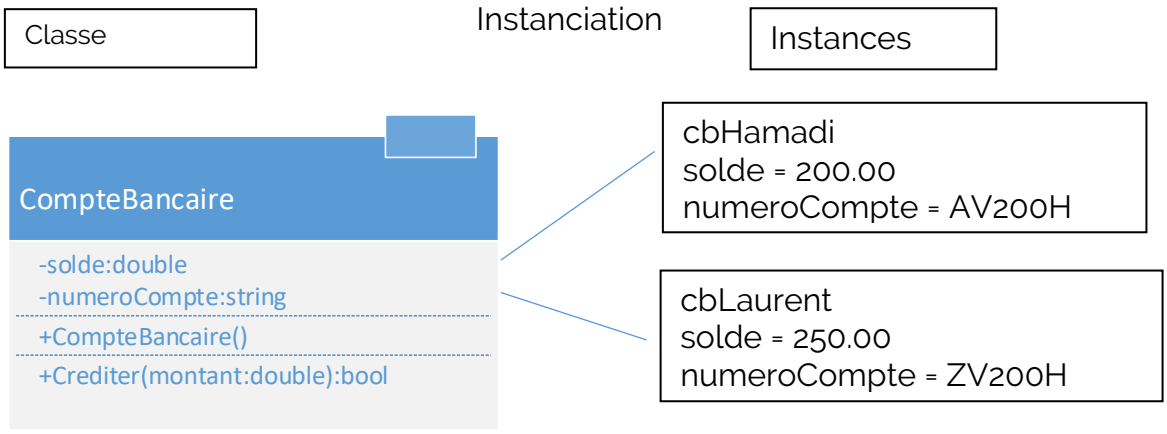
Un objet est une entité cohérente rassemblant des données et du code travaillant sur ses données. Une classe peut être vue comme un moule à partir duquel on peut créer des objets.
En fait on considère que la classe est la description de l'objet. Une classe décrit la structure interne de l'objet : les données qu'il regroupe, les actions qu'il est capable d'assurer sur ses données.
Considérons par exemple la modélisation d'un compte bancaire telle que présentée :



Et voici l'équivalent en JAVA:

```
public class CompteBancaire
{
    private string numeroCompte;

    private double solde;

    public compteBancaire(string nmCompte)
    {
        this.nmCompte = nmCompte;
    }

    public void crediter(double montant)
    {
        solde += montant;
    }
}
```

Dans ce modèle, un compte bancaire est représenté par une chaine de caractères (son numéro de compte) et son solde.
Ces données sont représentative d'un compte en particulier, autrement dit, chaque objet compte bancaire aura la propre copie de ses données. L'opération d'instanciation permet précisément de fournir des valeurs particulières pour chaque champ d'instance



Le même raisonnement s'applique directement aux méthodes. Prenons par exemple la méthode « Créditer ». Il est clair qu'elle peut s'appliquer individuellement à chaque compte bancaire pris comme entité séparée. En outre, cette méthode va clairement utiliser les attributs d'instance de l'objet auquel elle va s'appliquer c'est donc une méthode d'instance.

LE CONSTRUCTEUR

Si nous considérons en détail le processus permettant de créer un objet, nous nous apercevons que la première étape consiste à allouer de la mémoire pour le nouvel objet, hors cette étape n'est clairement pas du ressort d'un objet : seule la classe possède suffisamment d'informations pour la mener à bien : la création d'un objet est donc une méthode de classe.
En revanche, considérons la phase d'initialisation des attributs. Celle-ci s'applique à un objet bien précis : celui en cours de création. L'initialisation des attributs est donc une méthode d'instance. Nous aboutissons finalement au constat suivant : la création d'un nouvel objet est constituée de deux phases :

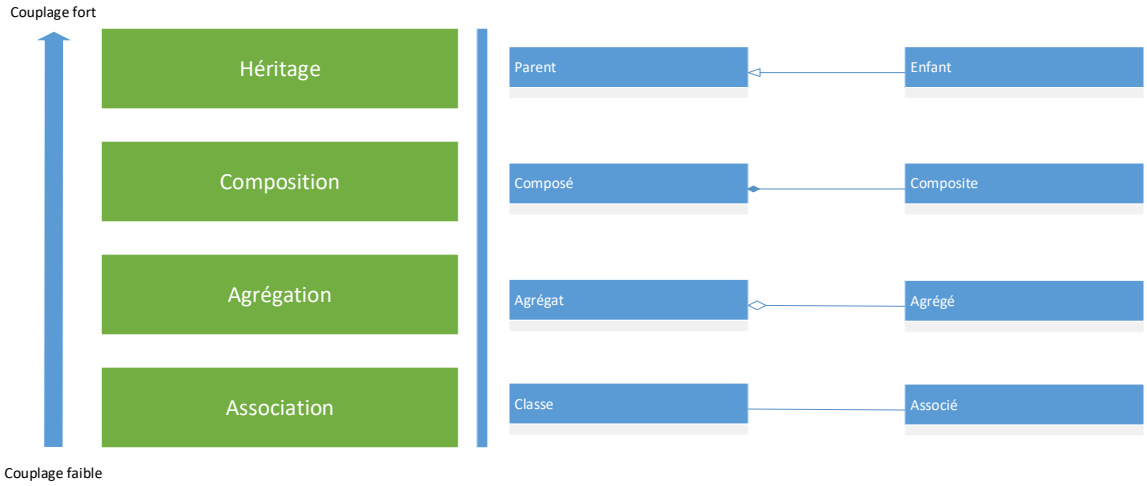
1. Une phase du ressort de la classe :
   Allouer de la mémoire pour le nouvel objet et lui fournir un contexte d'exécution minimaliste

2. Une phase du ressort de l'objet :
   Initialiser ses attributs d'instance

Si ces deux phases sont clairement séparées dans un langage tel que l'objective C, elles ne le sont pas en JAVA qui agglomèrent toute l'opération dans une méthode spéciale, ni vraiment méthode d'instance, ni méthode de classe : le constructeur.

LES RELATIONS INTER CLASSES

Voici pour commencer un schéma qui donnera une idée du niveau de couplage entre les différents types de relation.



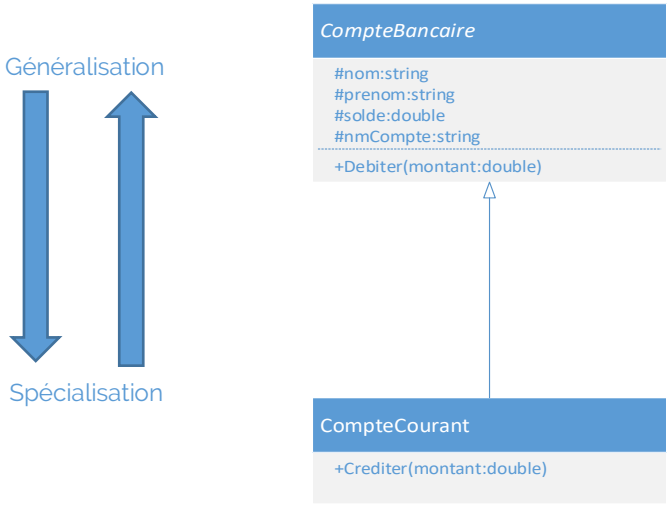
L'HERITAGE

L'héritage est le deuxième des trois principes fondamentaux du paradigme orienté objet (encapsulation, héritage, polymorphisme). Il est chargé de traduire le principe naturel de Généralisation/Spécialisation. En effet, la plupart des systèmes réels se prêtent à merveille à une classification hiérarchique des éléments qui les composent. La première idée à ce sujet est liée à l'entomologie et aux techniques de classification des insectes en fonction de divers critères. Expliquons nous d'abord sur le terme d'héritage. Il est basé sur l'idée qu'un objet spécialisé bénéficie ou hérite des caractéristiques de l'objet le plus général auquel il rajoute ses éléments propres.

En termes de concepts objets cela se traduit de la manière suivante :

- On associe une classe au concept le plus général, nous l'appellerons classe de base ou *classe mère* ou *super – classe*
- Pour chaque concept spécialisé, on dérive une classe du concept de base .La nouvelle classe est dite *classe dérivée* ou *classe fille* ou *sous classe*

L'héritage dénote une relation de généralisation/spécialisation, on peut traduire toute relation d'héritage par la phrase :
« La classe dérivée **est une** version spécialisée de sa classe de base »
On parle également de relation de type ***est-un*** pour traduire le principe de généralisation/spécialisation.
Considérons la super classe abstraite « compte bancaire » et sa classe fille « compte courant » :



Nous pourrons affirmer que c'est une relation de type «**est-un** », en effet un compte courant **est-un** compte bancaire, cette relation est facilement vérifiable.

Et la voici maintenant traduite java :

```
public  class CompteBancaire
{
        protected string nom;
        protected string prenom;
        protected string nmCompte;
        protected double solde;
}
public class CompteCourant extends CompteBancaire
{
        public void debiter(double montant)
        {
            throw new NotImplementedException();
        }

        public void crediter(double montant)
        {
            throw new NotImplementedException();
        }
}
```

ATTENTION A L'HERITAGE FONCTIONNEL

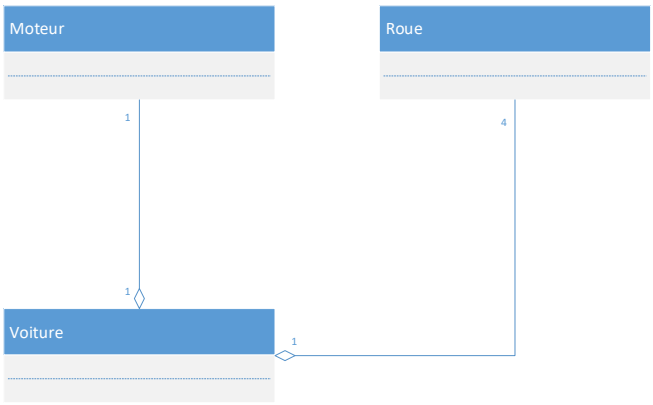
L'héritage n'est qu'une forme de couplage parmi d'autres, dès que la contrainte sur la nature des classes mères et dérivées n'est plus respectée, il faut absolument changer la manière de coupler les classes entre elles. Dériver les classes n'importe comment, sans justifier d'un quelconque lien conceptuel, dans le but de récupérer les fonctions d'une classe est un abus sévère du concept d'héritage. Cette méthode s'appelle : héritage fonctionnel et est à **proscrire**.

AGREGATION/COMPOSITION

Relation entre classes, indiquant que les instances d'une classe sont les composants d'une autre.

L'AGREGATION (COMPOSITION FAIBLE)

L'agrégation est un autre type de relation entre deux classes qui traduit cette fois les relations «est composé de… » Ou bien «possède … « ou encore « a … ».Par exemple dans un système mécanique, on pourrait considérer que la classe voiture est composé d'une instance de la classe moteur, quatre ou cinq(!) Instances de la classe roue. L'instanciation passe nécessairement par l'utilisation d'attributs qui sont eux-mêmes des objets. L'une des caractéristiques principale de l'agrégation est sa cardinalité. Considérons par exemple l'agrégation de roues par une voiture. Une voiture possède exactement 4 roues (je laisse la roue de secours) et chaque roue ne peut pas être possédée par plus d'une voiture. La cardinalité de l'agrégation est donc de 1 côté agrégateur et de 4 du côté agrégé.
Voici ce concept traduit en UML :



Maintenant en JAVA :

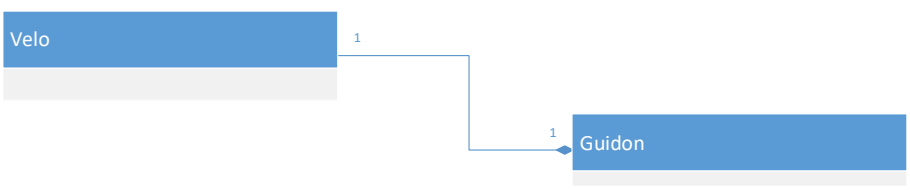
```
public class Moteur {}
public class Roue {}
public class Voiture
{
      private Moteur moteur= new Moteur();
      private Roue[] roues = new Roue[3];

      public Voiture(Moteur  moteur, Roue[] roues)
      {
         this.moteur = moteur;
         this.roues = roues ;
      }
}
```

La notation UML utilise une flèche dont la pointe est un losange pour représenter l'agrégation. Le losange est du côté de l'agrégateur (de la classe composée). Les cardinalités sont indiquées :

- à côté du losange pour la cardinalité de l'agrégateur (sur le schéma : une roue appartient à une seule voiture)
- à côté du trait pour la cardinalité de l'agrégé (sur le schéma : une voiture possède exactement 4 roues)

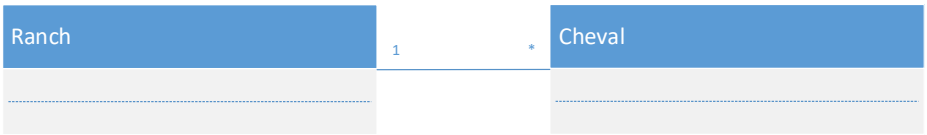
LA COMPOSITION (AGREGATION FORTE)

Les objets agrégés ont une durée de vie dépendante de celle de l'agrégat, La notation UML utilise une flèche dont la pointe est un losange (plein).



L'ASSOCIATION (LIEN FAIBLE)

L'association est la troisième grande forme de relation que nous allons considérer après celle de l'héritage et de l'agrégation. Elle consiste à une simple relation de collaboration entre deux objets. Chaque objet existe de manière totalement indépendante, et n'a pas forcément besoin de l'autre pour "vivre". La relation entre un Ranch et ses chevaux est une relation d'association.



Et en JAVA :

```
public class Ranch
{
      public List<Cheval> Chevaux ;

}
```

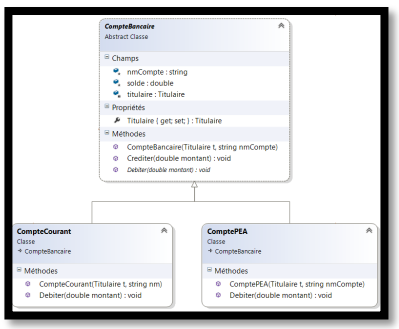
LES TYPES DE CLASSES

CLASSES ABSTRAITES

De la même façon qu'il est possible d'empêcher quelqu'un d'étendre une classe, ou de surcharger une méthode, il est possible de forcer l'extension ou la surcharge, en utilisant le mot-clé abstract. Formellement une classe abstraite n'est pas diff érente d'une classe normale. Simplement, elle est déclarée en ajoutant le mot-clé abstract.

On peut déclarer autant de méthodes abstraites que l'on veut dans une classe abstraite, y compris ne pas en déclarer du tout. Une méthode abstraite, comme dans notre exemple, ne comporte qu'une déclaration de signature ; elle n'a pas de corps. On peut alors se poser la question : comment fait-on pour exécuter une telle méthode ? La réponse est simple : on ne peut pas l'exécuter en l'état.

Une classe abstraite ne peut pas être instanciée. Il est nécessaire de créer une classe concrète (c'est-à-dire non abstraite !) qui l'étende, et d'instancier cette classe concrète. Toute classe concrète qui étend une classe abstraite doit fournir un corps de méthode (on parle alors d'implémentation, un beau mot de la langue française...) pour toutes les méthodes abstraites de cette classe abstraite. Dans notre exemple, on peut alors écrire le code de l'exemple suivant.



```
public abstract class CompteBancaire
{
        protected  Titulaire titulaire;
        protected string nmCompte;
        protected double solde;


        public compteBancaire(Titulaire t, string nmCompte)
        {
            this.titulaire = t;
            this.nmCompte = nmCompte;
        }

        public void crediter(double montant)
        {
            solde += montant;
        }

        public abstract void debiter(double montant);

}
public class CompteCourant extends CompteBancaire
{

        public compteCourant(Titulaire t, string nm)

        {
            super(t, nm) ;
        }

        @Override
        public  void debiter(double montant)
        {
            solde -= montant;
        }
}
```

CLASSES CONCRETES

Une classe concrète n'est rien d'autres qu'une classe normale.

CLASSES « FINAL »

Appliqué à une classe, le modificateur **final** empêche d'autres classes d'en hériter.
Dans l'exemple suivant, la classe B hérite de la classe A, mais aucune classe ne peut hériter de la classe B.

```
public final class ClasseA
{
        public  void display()
        {
            System.out.println("Methode display ClasseA");
        }
}

public class ClasseB : ClasseA
{
        @Override
        public void display()
        {
            super.display();
            System.out.println("Methode display Clas
        }
}
```

Impossible

CLASSES STATIQUES

Une classe statique est rigoureusement identique à une classe non statique, à ceci près qu'une classe statique ne peut pas être instanciée ! En effet, vous ne pourrez pas utiliser le mot clé new pour instancier votre variable. L'utilité de créer un champ statique est de pouvoir accéder à cette valeur unique depuis n'importe quel endroit.

Par exemple, si vous avez une classe statique nommée MaClasse qui a une méthode publique nommée Foo, vous appelez la méthode comme illustré dans l'exemple suivant :

```
MaClasse.foo();
```

Et sa déclaration :

```
public class MaClasse
{
    public static void foo()
    {
    }
}
```

LE POLYMORPHISME

Classification de CARDELLI

LE POLYMORPHISME DE TRAITEMENT AD HOC(*OVERLOADING*)

Lorsqu'on définit une classe, on peut y définir plusieurs méthodes qui portent le même nom. La seule contrainte est que la liste des types des paramètres formels doit être différente. On parle de *surcharge de méthode* (method overloading en anglais). Il n'y a aucune contrainte sur les différents modificateurs ou sur la valeur de retour. Le mécanisme de surcharge s'applique également aux constructeurs.
Lors de la création d'un nouvel objet de la classe Utilisateur, le bon constructeur sera appelé en fonction des types de la liste des paramètres. Si on créé un objet en faisant new Utilisateur();, comme la liste des paramètres est vide, c'est le premier constructeur qui sera exécuté. Mais si on fait par exemple new Utilisateur («CHEIKH»);, c'est le second constructeur qui sera exécuté puisque la chaine «CHEIKH » est de type string, il faut donc un constructeur qui prenne un paramètre de type string.

RESOLUTION DE SURCHARGE

C'est lors de la compilation que la méthode surchargée qui sera appelée est choisie. Ce choix est fait par le compilateur en fonction des types des paramètres réels. Le compilateur tout d'abord compter le nombre de paramètres réels puis va construire la liste des types des paramètres réels. Il recherche ensuite une méthode qui corresponde à cette liste.

```
public class Utilisateur
{
        private string nom;

        public Utilisateur()
        { }

        public Utilisateur(string nom)
        {
            this.nom = nom;
        }

        public string GeneratePassword(string salt)
        {
            return new Random().nextInt(999999) + salt;
        }

        public string GeneratePassword()
        {
            return new Random().nextInt(999).toString();
        }
}
```

LE POLYMORPHISME DE DONNEES(UNIVERSEL)

LE POLYMORPHISME D'INCLUSION (REDEFINITION OU OVERRIDE)

Ce type est lié à l'héritage et plus exactement au sous-typage (ordre sur les types), il s'agit d'utiliser comme argument un objet de classe B qui hérite de la classe A à la place d'un objet de la classe A. Il y a des subtilités (cf. variance et contra variance).En C# ce sont les mots clés override et virtual qui matérialise le concept.

```
    public class ClasseA
    {
        public  void Display()
        {
            System.out.println("Methode display ClasseA");
        }
    }

    public class ClasseB : ClasseA
    {
        @Override
        public  void display()
        {
            super.Display();
            System.out.println("Methode display ClasseB");
        }
    }
```

LE POLYMORPHISME PARAMETRIQUE (GENERICITE OU TEMPLATE)

Les génériques sont des types et méthodes paramétrés. Chaque paramètre est une marque de réservation destinée à accueillir un type qui n'est pas encore spécifié. Le comportement polymorphique du type ou de la méthode générique est véhiculé par les paramètres.

C'est ce que l'on appelle polymorphisme paramétrique. Il y a une seule implémentation de l'algorithme que l'on transforme par les paramètres.

```
    public class Pile<T>
    {
        private T[] elements;
        private int nbElements;
        public Pile(T n)
        {
            this.elements = new T[n];
            this.nbElements = 0;
        }
        public T GetElement(int n)
        {
            return this.elements[n];
        }
    }
Implantation :
  Pile<int> pileEntiers = new Pile<int>();
  Pile<String> pileChaines = new Pile<String>();
```