Projet de

Sémantique et TDL

–

Compilation du langage Micro-Java

Réalisé par

Dalbert Jérôme
Foulfoin Anthony
Pastor Harper Javier

Table des matières

1. La table des symboles

1.1 Conception

Nous sommes partis sur deux constatations simples :

- La table des symboles associe une information à un symbole.
- Dans notre projet, les informations que l'on veut stocker sont des classes, des attributs, des méthodes et des variables locales.

La création des classes pour la gestion de la table des symboles se fait alors naturellement.

Nous avons repris la classe TDS vue en TP en lui ajoutant le type générique T, de manière à éviter de faire des `match` ou des `cast` systématiquement dans le code.

Pour signifier qu'à un symbole, on associe une information, nous avons créée l'interface marqueur Info. Toute information destinée à être stockée dans la table doit l'implémenter. Au final, la classe représentant la table des symboles paramétrée est donc TDS<T extends Info>.

Une classe est principalement caractérisée par des attributs et des méthodes.

Nous avons donc une classe InfoClasse qui possède une TDS d'InfoAttributs et une TDS d'InfoMethodes : ainsi nous pouvons rapidement retrouver les informations associées au nom d'un attribut/méthode.

Héritage oblige, l'InfoClasse possède une InfoClasse mère. Celle-ci possède également un seul constructeur (notre compilateur n'accepte pas la surcharge) et un déplacement de sa table des virtuelles (voir la génération de code partie 3).

Enfin, nous considérons qu'une classe est un type. InfoClasse hérite donc de Type.

Soit par exemple l'instruction : `Point p;`

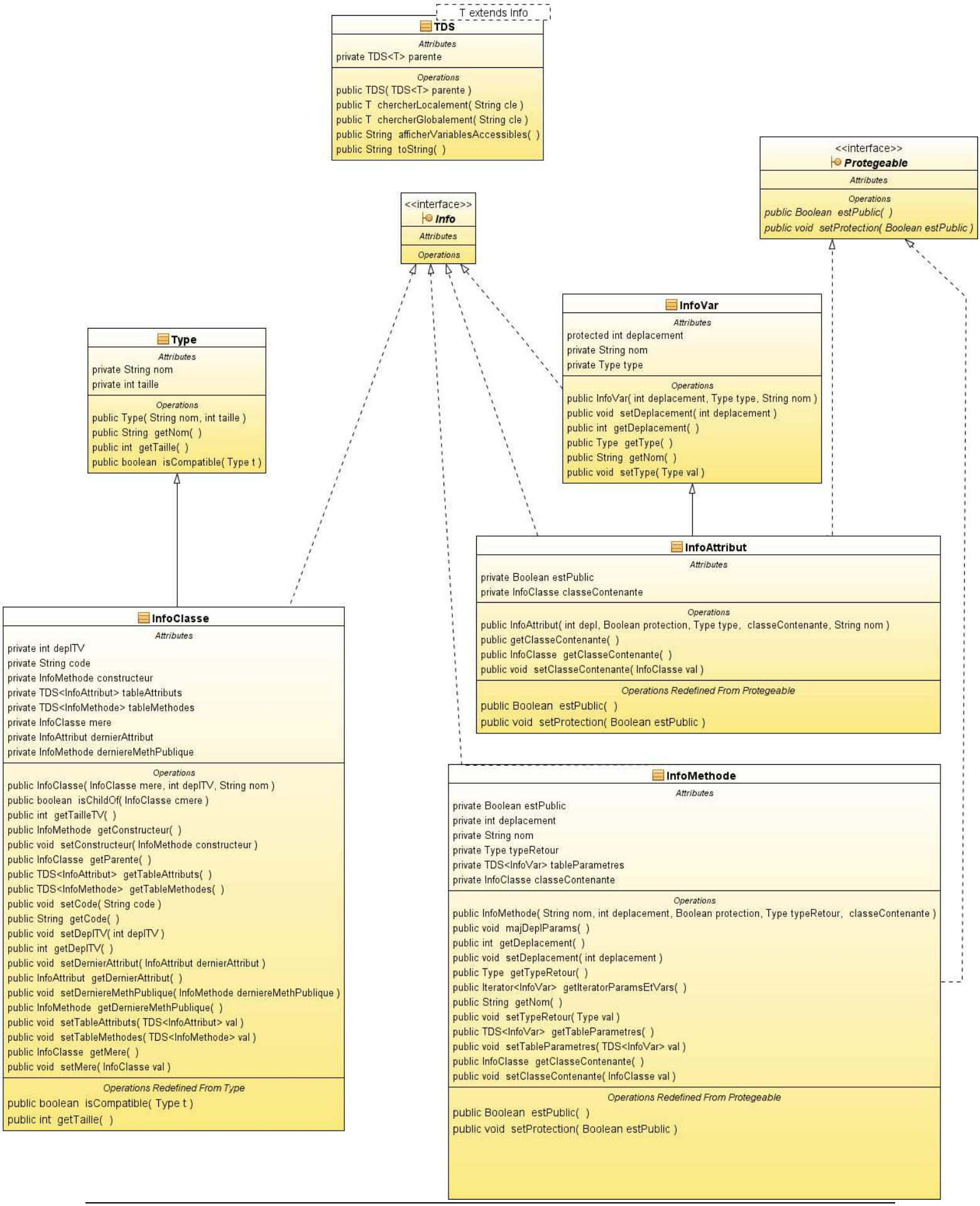
L'objet p est de type l'InfoClasse de nom "Point".

Les InfoAttributs et InfoMethodes ont une protection (publique ou privée) et implémentent donc l'interface Protégeable. Si l'on souhaite faire évoluer le langage et protéger une classe comme dans le vrai langage Java (public class …), il suffira alors à InfoClasse d'implémenter Protégeable.

Un InfoAttribut n'est qu'une variable possédant une protection, il hérite donc de la classe représentant une variable. Sachant qu'une variable peut être stockée dans une TDS (notamment les variables locales d'un bloc), la classe correspondante se nomme InfoVar.

Une InfoMethode possède un type de retour, des paramètres stockés sous forme d'une TDS<InfoVar>, et un déplacement par rapport à la table des virtuelles de sa classe.

Le diagramme de classe d'héritage du paquetage mjc.tds se trouve sur la page suivante.


T extends Info
TDS
Attributes
private TDS<T> parente
Operations
public TDS(TDS<T> parente)
public T chercherLocalement(String cle)
public T chercherGlobalement(String cle)
public String afficherVariablesAccessibles( )
public String toString( )
<<interface>>
Protegeable
Attributes
Operations
public Boolean estPublic( )
public void setProtection(Boolean estPublic)
<<interface>>
Info
Attributes
Operations
InfoVar
Attributes
protected int deplacement
private String nom
private Type type
Operations
public InfoVar(int deplacement, Type type, String nom)
public void setDeplacement(int deplacement)
public int getDeplacement( )
public Type getType( )
public String getNom( )
public void setType(Type val)
Type
Attributes
private String nom
private int taille
Operations
public Type(String nom, int taille)
public String getNom( )
public int getTaille( )
public boolean isCompatible(Type t)
InfoAttribut
Attributes
private Boolean estPublic
private InfoClasse classeContenante
Operations
public InfoAttribut(int depl, Boolean protection, Type type, classeContenante, String nom)
public getClasseContenante( )
public InfoClasse getClasseContenante( )
public void setClasseContenante(InfoClasse val)
Operations Redefined From Protegeable
public Boolean estPublic( )
public void setProtection(Boolean estPublic)
InfoClasse
Attributes
private int deplTV
private String code
private InfoMethode constructeur
private TDS<InfoAttribut> tableAttributs
private TDS<InfoMethode> tableMethodes
private InfoClasse mere
private InfoAttribut dernierAttribut
private InfoMethode dernieremethPublique
Operations
public InfoClasse(InfoClasse mere, int deplTV, String nom)
public boolean isChildOf(InfoClasse cmere)
public int getTailleTV( )
public InfoMethode getConstructeur( )
public void setConstructeur(InfoMethode constructeur)
public InfoClasse getParente( )
public TDS<InfoAttribut> getTableAttributs( )
public TDS<InfoMethode> getTableMethodes( )
public void setCode(String code)
public String getCode( )
public void setDeplTV(int deplTV)
public int getDeplTV( )
public void setDernierAttribut(InfoAttribut dernierAttribut)
public InfoAttribut getDernierAttribut( )
public void setDerniereMethPublique(InfoMethode dernieremethPublique)
public InfoMethode getDerniereMethPublique( )
public void setTableAttributs(TDS<InfoAttribut> val)
public void setTableMethodes(TDS<InfoMethode> val)
public InfoClasse getMere( )
public void setMere(InfoClasse val)
Operations Redefined From Type
public boolean isCompatible(Type t)
public int getTaille( )
InfoMethode
Attributes
private Boolean estPublic
private int deplacement
private String nom
private Type typeRetour
private TDS<InfoVar> tableParametres
private InfoClasse classeContenante
Operations
public InfoMethode(String nom, int deplacement, Boolean protection, Type typeRetour, classeContenante)
public void majDeplParams( )
public int getDeplacement( )
public void setDeplacement(int deplacement)
public Type getTypeRetour( )
public Iterator<InfoVar> getIteratorParamsEtVars( )
public String getNom( )
public void setTypeRetour(Type val)
public TDS<InfoVar> getTableParametres( )
public void setTableParametres(TDS<InfoVar> val)
public InfoClasse getClasseContenante( )
public void setClasseContenante(InfoClasse val)
Operations Redefined From Protegeable
public Boolean estPublic( )
public void setProtection(Boolean estPublic)

1.2 Gestion des tables des symboles

1.2.1 La table des Classes

Lors de l'analyse syntaxique, il faut pouvoir retrouver ou ajouter une classe à n'importe quel moment : aussi bien lors d'une instruction d'un bloc que lors de la déclaration d'une classe/méthode/attribut.

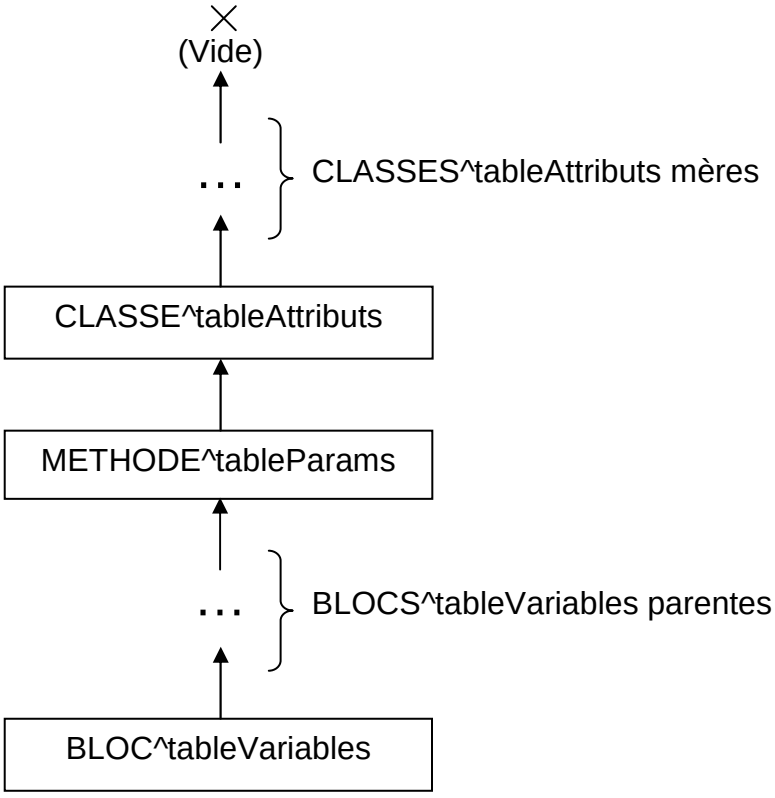
Pour ce faire, les règles de production du fichier egg ont accès à un attribut tdc (de type TDS<InfoClasse>) hérité depuis la règle racine

```
PROGRAMME -> IMPORTS DEFCLASSE;
```

Ainsi, la table des classes est accessible n'importe où dans le programme.

1.2.2 Hiérarchie des tables

Les noms en majuscule représentent des entités logiques (non terminaux EGG ou classes Java). Une flèche partant d'une table 1 à une table 2 signifie que la table 1 a pour parente la table 2.



Si on se trouve dans un bloc et qu'on fait une recherche globale d'un symbole dans la tableVariables, on cherchera d'abord dans la table du bloc, puis éventuellement dans celles des blocs parents, puis éventuellement dans celle des paramètres de la méthode contenante, puis éventuellement dans celle des attributs de la classe contenante, puis éventuellement dans celles des classes mères.

1.2.3 Gestion de la classe courante

Dans la règle de production de déclaration d'une classe, on crée la classe et on l'insère dans la table des classes avant d'analyser son contenu. Il ne faut pas attendre d'avoir analysé toutes les informations de la classe pour la créer.

En effet, le contenu d'une classe C peut comporter des attributs de classe C ou des méthodes qui utilisent C. En créant et en insérant la classe dès le début, si on tombe sur une référence à la classe C dans le contenu, on trouvera cette classe dans la table des symboles.

Remarque : Après l'analyse de la classe courante, si on n'a pas trouvé de constructeur, on en ajoute un par défaut. On reproduit ainsi le comportement de Java.

1.2.4 Déplacement des Info

Il se trouve que toutes les classes implémentant Info ont un déplacement (ça pourrait ne pas être le cas, si on gérait les typedef par exemple). Mais ce déplacement a un sens différent selon la classe et le contexte.

- Le déplacement d'une variable locale (InfoVar) se fait par rapport à LB qui est le registre utilisé lors de l'appel de procédure. En effet, toutes les instructions du programme se trouvent dans des méthodes, qui seront traduites par des procédures en TAM. Le déplacement de la première variable locale est 3, car ceux de 0 à 2 sont réservés.
- Le déplacement d'un paramètre (InfoVar) de méthode se fait aussi par rapport à LB, mais il est négatif. C'est comme ça que TAM gère les paramètres de ses procédures.
- Le déplacement d'un InfoAttribut se fait par rapport à l'adresse (dans le tas) de la classe instanciée.
- Le déplacement d'une InfoMéthode se fait par rapport à la position de début (dans la pile SB) de la table des virtuelles de la classe qui contient cette InfoMéthode.
- Le déplacementTV d'une InfoClasse est le déplacement de la table des virtuelles de l'InfoClasse. Il se fait par rapport à la position de début (dans la pile SB) de toutes les tables virtuelles.

1.2.5 Importation de classes

Le fichier egg de ce projet ne permet d'analyser qu'une seule classe par défaut. Pour résoudre ce problème, l'idée est d'appeler la méthode main de la classe principale MJC pour chaque classe à analyser.

Soit X le nom de la classe à analyser. MJC.main(X) analyse X, ce qui fait qu'on obtient un InfoClasse C. A partir de là, il faut enregistrer C pour le communiquer aux autres appels de MJC.main. Nous avons envisagé deux solutions :

- Enregistrer C sur le disque dur, en utilisant la sérialisation.

  L'importation de X se fait alors en appelant MJC.main(X) puis en chargeant dans la table des classes tdc le fichier résultat correspondant.

- Enregistrer C dans la table des classes tdc (il faut que tdc soit un attribut statique de MJC).

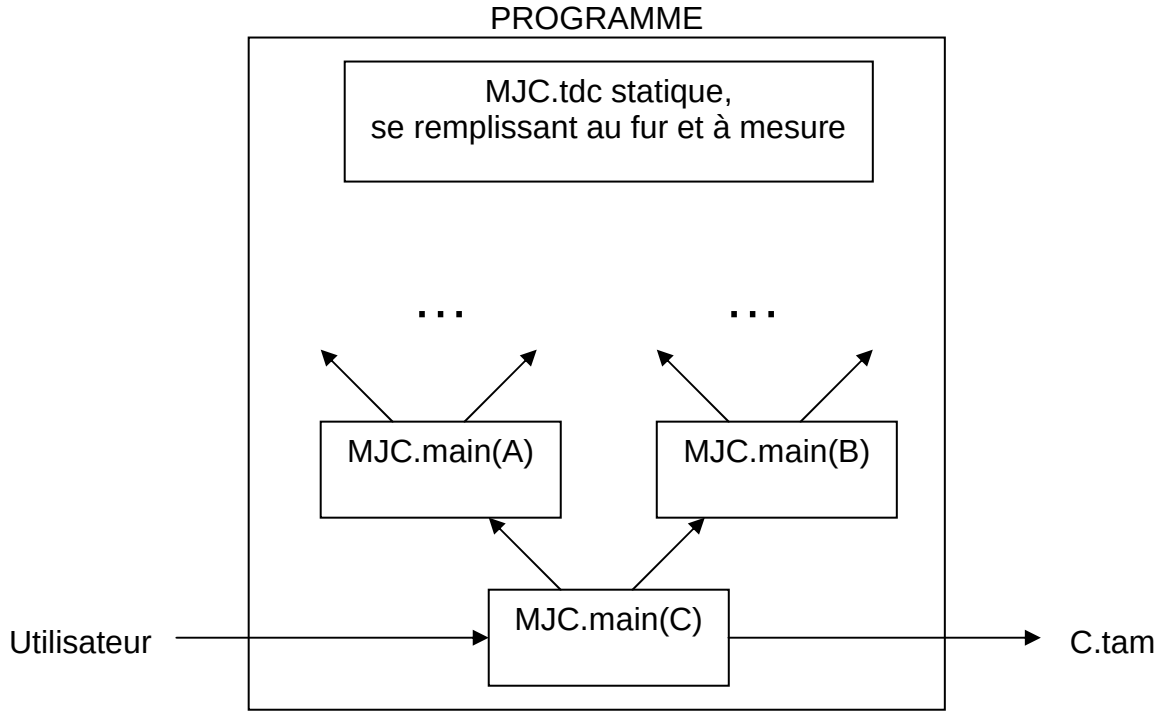
  L'importation de X se fait alors seulement en appelant MJC.main(X). Le prochain appel à MJC.main aura accès à ce X car MJC.tdc est statique.

Nous avons retenu la deuxième solution, plus simple à mettre en œuvre et plus rapide car on ne fait aucun appel disque.

Soit ci-dessous le code Micro-Java que l'utilisateur veut compiler:

```
Import A ;     // A peut importer d'autres classes
Import B ;     // B peut importer d'autres classes
Class C { … }
```

Voici un schéma de l'exécution du programme (on remarque la récursivité) :



2. Contrôle de type

Une fois la table des symboles remplie, nous pouvons utiliser les informations qu'elle contient afin d'effectuer des vérifications au sein du programme. Ces contrôles vérifient l'existence des variables utilisées dans le programme, l'existence des types utilisés, et que chaque expression possède bien le type qui était attendu.

Pour cela, chaque expression possède un type qui est synthétisé et qui remonte petit à petit permettant ainsi d'effectuer des vérifications portant sur ce dernier au sein d'autres expressions ou d'instructions, comme par exemple la condition d'un if qui doit être de type booléen.

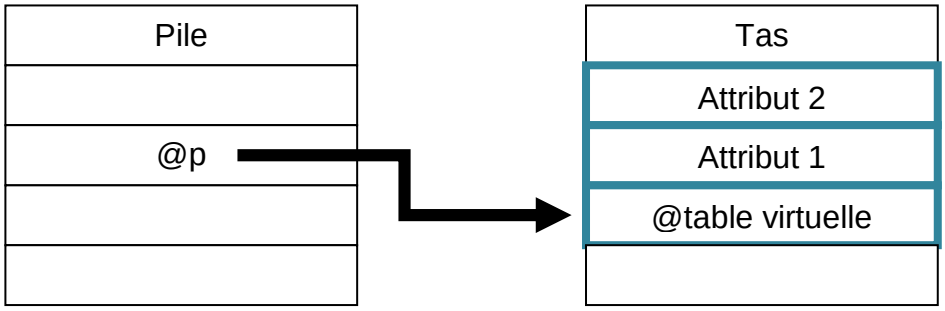
Cette étape comporte également une vérification de l'accessibilité des attributs et des méthodes en fonction de leur visibilité. Les attributs ou les méthodes privées ne peuvent être utilisés qu'au sein de la classe courante, cette vérification est donc effectuée, et génère une erreur si le programme souhaite accéder à un élément privé d'une autre classe.

3. Génération de code

Une fois le remplissage de la table des symboles et le contrôle de type accomplis, s'est alors posé le problème de la génération de code. Plusieurs difficultés majeures sont alors apparues concernant la conception de cette partie, notamment le problème de la liaison tardive.

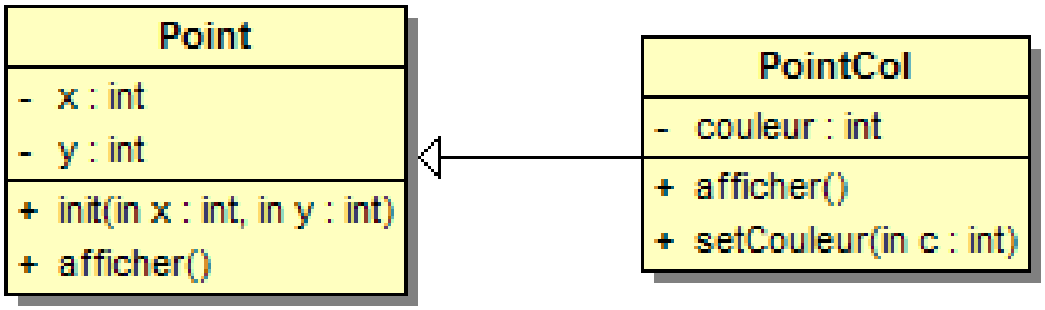
3.1 Représentation des objets en mémoire

Un objet est caractérisé par ses attributs et son type dynamique (représenté ici par un pointeur sur une table des méthodes virtuelle). Si un objet p possède n attributs, nous créons donc une zone mémoire de taille n+1 dans le tas. La première adresse de cette zone contient l'adresse de la table des méthodes virtuelle (voir plus loin), au-dessus de la quelle nous empilons les valeurs des attributs de l'objet.



3.2 Gestion de la liaison tardive

Pour comprendre plus simplement le problème, voici un petit exemple :



Nous disposons de deux classes Point et PointCol. PointCol redéfinit une méthode de Point, et définit une méthode supplémentaire.

Le problème apparaît lors de l'instanciation d'un nouvel objet :

```
Point p = new PointCol() ;
```

Ici le point p possède deux types : un type statique, et un type dynamique. Le type dynamique est un sous type du type statique, et n'est connu qu'au moment de l'exécution du programme.

Si nous souhaitons exécuter l'instruction : p.afficher(), la méthode qui doit être exécutée est celle du type dynamique, que nous ne connaissons pas lors de la compilation. Viennent alors quelques constatations :

- Le choix devra se faire dynamiquement pendant l'exécution, un objet doit donc « mémoriser » une information concernant son type dynamique lorsqu'il est instancié.
- Cette information doit nous permettre de trouver quelle méthode doit être exécutée.
- Le type dynamique possède au moins les méthodes publiques du type statique.

Partant de cette constatation, nous avons choisi de créer pour chaque classe une table au début de la pile d'exécution du programme listant l'adresse de ses méthodes. Cette table sera appelée table des méthodes virtuelles (TV).

Chaque méthode possède un numéro qui correspond au déplacement à effectuer par rapport à la base de la TV de sa classe pour trouver son adresse.

Dans le cas où une classe A hérite de B, celle-ci possède la même TV que B, en dehors des méthodes redéfinies et des nouvelles méthodes.

Si A redéfinit une méthode de B, celles-ci possèderont toutes les deux le même déplacement, et seront donc placées au même endroit dans leurs TV respectives. Les nouvelles méthodes publiques seront ajoutées à la suite de la TV.

@Point_init depl : 0 @Point_afficher depl : 1	TV de Point Depl de la classe : 0
@Point_init depl : 0 @PointCol_afficher depl : 1 @PointCol_setCouleur depl : 2	TV de PointCol Depl de la classe : 2

Ainsi pour retrouver l'adresse d'une méthode, nous avons besoin de connaitre le début de la TV de la classe associée au type dynamique de l'objet, auquel il faut ajouter le déplacement de la méthode au sein de cette table, obtenu grâce au type statique que nous connaissons au moment de la compilation.

Il nous reste donc à obtenir l'adresse de la TV associée à un objet. Pour ce faire, si un objet possède n attributs, nous lui créons un espace mémoire de taille n+1. La case supplémentaire (la première de cet espace mémoire) permet de stocker l'adresse (le déplacement de la classe) de sa TV, qui est donc mémorisée lors de l'instanciation de cet objet.

3.3 Représentation de l'héritage

L'héritage est présent en deux endroits :

- l'héritage des méthodes, qui est représenté notamment grâce à la TV.
- Au niveau des attributs. Ainsi en plus de réserver de la mémoire dans le tas pour les attributs de la classe, nous réservons également de la mémoire pour les attributs publics et privés des superclasses afin de préserver le système d'héritage. Chaque attribut possède un déplacement, qui correspond à son adresse dans le tas par rapport à la base de l'espace mémoire de l'objet. Considérant le système d'héritage, une classe commence la numérotation de ses attributs selon le dernier attribut de la classe mère.

3.4 Gestion de l'objet courant « this »

Afin qu'une méthode puisse consulter, ou modifier les attributs de l'objet pour lequel elle s'exécute, il est nécessaire que celle-ci possède un pointeur sur la zone mémoire de l'objet courant.

Pour ce faire, lors d'un appel de méthode, nous empilons l'adresse de l'objet concerné devant les paramètres.

Ainsi si nous exécutons p.set(x), nous empilons p, puis le paramètre x avant d'appeler la méthode.

A tout moment, une méthode peut ainsi obtenir un pointeur sur l'objet courant en chargeant le contenu de l'adresse –(n+1)[LB] où n représente le nombre de paramètres de la méthode.

3.5 Accès à une méthode

Lorsque l'on souhaite invoquer une méthode, deux cas sont possibles :

- On invoque une méthode de l'objet courant, par exemple methodeInutile()
- On invoque une méthode associée à un autre objet, par exemple p.methodeInutile().

3.5.1 Méthode « locale »

L'accès à une méthode « locale » peut être généré de manière statique lors de la compilation. On charge le pointeur de l'objet courant, puis les paramètres, puis on charge l'adresse de la méthode directement grâce à son étiquette avant de l'invoquer.

3.5.2 Méthode « distante »

L'invocation d'une telle méthode est plus complexe car celle-ci fait intervenir la liaison tardive. Dans le cas où nous invoquons par exemple p.methode(x,y) :

- Nous empilons la valeur du pointeur p qui sera l'objet courant de la méthode
- Nous empilons les paramètres
- Nous obtenons l'adresse de la TV en chargeant le contenu de la case pointée par p.
- Nous incrémentons cette valeur avec le déplacement associé à la méthode
- Nous chargeons la valeur pointée par cette adresse
- Nous avons l'adresse de la méthode recherchée, il ne reste plus qu'à l'invoquer.

3.6 Accès à un attribut

Afin d'accéder à un attribut, nous avons besoin de son déplacement, et d'un pointeur sur son objet.

Si l'on souhaite accéder à un attribut de l'objet courant, on charge le pointeur sur l'objet courant qui se trouve en –(n+1)[LB]. Sinon on charge le pointeur de l'objet concerne.

Puis on ajoute à cette valeur le déplacement de l'attribut, que l'on charge ensuite dans la pile.

3.7 Gestion des pointeurs null

Un pointeur null est considéré comme étant un pointeur ayant une adresse égale à 0. Il est dès lors possible :

- d'affecter la valeur null à un pointeur : Point p = null ;
- de vérifier si un pointeur est null : if (p==null) …

Lorsqu'un objet est déclaré sans initialisation, par exemple Point p, celui-ci prend automatiquement la valeur null.