JavaCC-AST Eclipse

INF 1018 – Analyse de programme
Automne 2020

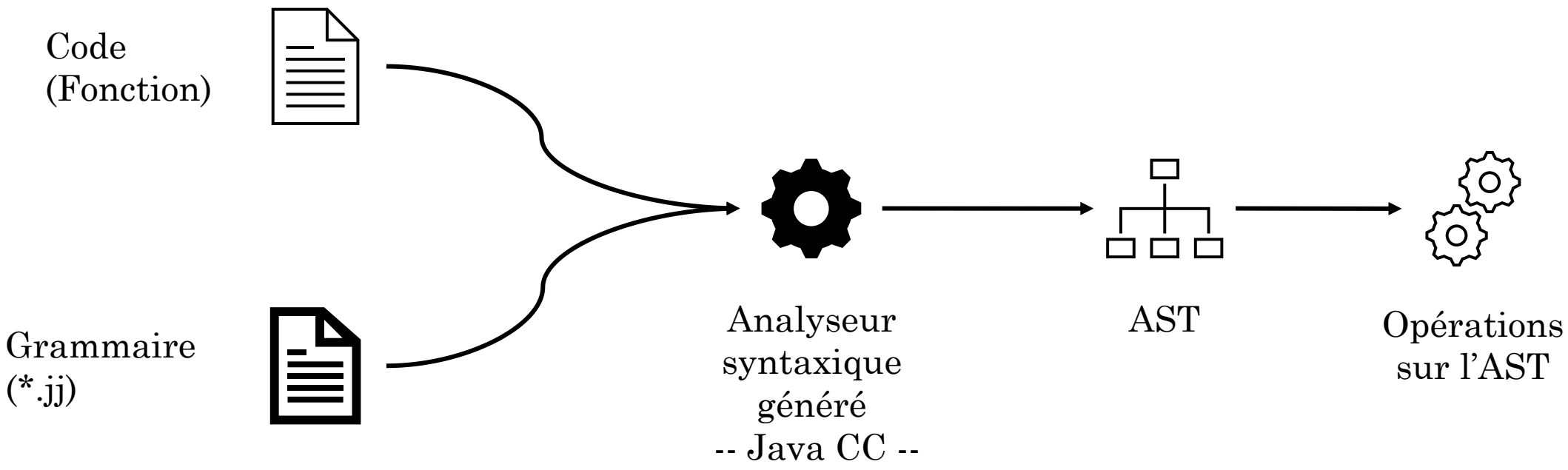
Plan

- Outil JavaCC
- Abstract Syntax Tree (AST)
  - Structurer un AST
- AST d'Eclipse
  - Installer Eclipse PDE
  - Créer un projet
  - Définir un visiteur
- ASTView (Complément)
  - Visualiser les AST dans Eclipse

Java Compiler Compiler (Java CC)

- Générateur de parser (analyseur syntaxique)
- Outil comparable à yacc, antlr…
- Supporte les grammaires LL(k) et utilise la notation EBNF
- Utilise la récursivité pour déclarer ses règles
- On peut insérer du code dans la grammaire pour permettre de créer un AST

Java CC


Code
(Fonction)
Grammaire
(*.jj)
Analyseur
syntaxique
généré
-- Java CC --
AST
Opérations
sur l'AST

Java CC

Un fichier de grammaire (*.jj) est divisé en 3 parties toujours dans le même ordre :

1) Options
2) Classe Parser (vous pouvez la renommer)
3) La définition de la grammaire

Options

Définit le comportement du parser.

Options principales :

Option	Effet	Défaut
LOOKAHEAD	Nombre de token à regarder avant de prendre une décision	1
STATIC	Est-ce que les méthodes du parser généré sont statiques ?	True
IGNORE_CASE	Est-ce que le langage est insensible à la casse ?	False
BUILD_PARSER	Est-ce que le parser doit être généré ?	True
OUTPUT_DIRECTORY	Répertoire de sortie des fichiers générés	Le répertoire courant
DEBUG_PARSER	Affiche toutes les opérations du parser (utile si vous modifiez la grammaire)	False

Options

Exemple de bloc d'options

```
options
{
  static = true;
  debug_parser = false;
}
```

Classe Parser

Définit la classe java qui représentera le parser.

Tout le code défini dans le fichier (y compris dans la grammaire et les options) sera du code envoyé dans cette classe.

Classe Parser

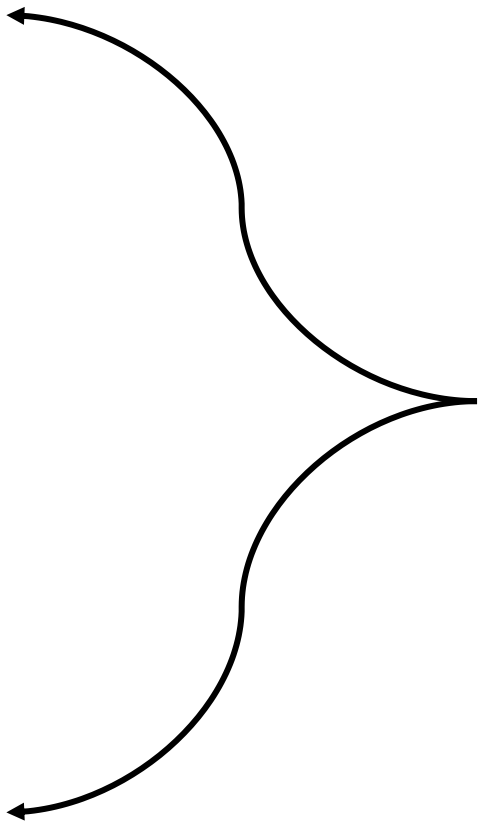
```
PARSER_BEGIN(Parser)
package parser;

import java.io.*;
...

public class Parser
{
    ...

}

PARSER_END(Parser)
```



Définit les limites du code Java

Spécification de la grammaire

La grammaire se divise en 2 zones :

1) Définition des tokens
   ➔ Token sont des mots ou des séquences capturés comme un mot-clé ou un identifiant

2) Règle de dérivation
   ➔ Structure de la grammaire
   ➔Ressemble à des méthodes de classe

Token

```
TOKEN : /* OPERATEURS */
{
  < PLUS : "+" >
| < MINUS : "-" >
}
```

Chaque Token possède un nom et un symbole (string) qui le représente.

On utilise le nom du Token (par exemple <PLUS>) pour y faire référence dans la grammaire

Token

On peut former plusieurs blocs de Token

Les Token peuvent aussi représenter des classes de caractères

```
TOKEN :/* LITTERAUX */
{
  < #DIGIT: ["0"-"9"] >
| < #LETTER : ["A"-"Z", "a" - "z"] >
}
```

Quantificateurs et logique de JavaCC

Pour exprimer un choix entre 2 options, on utilise la barre verticale « | »

On peut utiliser 3 quantificateurs pour exprimer des répétitions ou le caractère optionnel d'une expression:

- ? : 0 ou 1 fois
- * : 0, 1 ou plusieurs fois
- + : au moins une fois

Les parenthèses permettent de créer des groupes et d'affirmer la priorité.

Quantificateurs et logique de JavaCC

```
TOKEN :/* LITTERAUX */
{
  < #DIGIT: ["0"-"9"] >
| < #LETTER : ["A"-"Z", "a" - "z"] >
| < INTEGER : ("-")? ( < DIGIT > )+ >
| < DECIMAL :
        ("-")? (< DIGIT >)+ "." (< DIGIT >)*
        |  ("-")? (< DIGIT >)* "." (< DIGIT >)+
        >
| < IDENTIFIER : < LETTER > (< LETTER > | < DIGIT >)* >
}
```

SKIP, MORE et SPECIAL_TOKEN

Il s'agit de Token avec des comportements particuliers

- SKIP : ignorent les caractères spécifiés
- MORE : collecte des caractères dans le même token (pour ignorer par exemple)
- SPECIAL_TOKEN : indique la fin de la collecte

Utilisés par exemple pour définir les commentaires

Règles de la grammaire

Ressemble à une méthode dans Java (à la compilation, elles deviennent des méthodes dans la classe Parser).

```
void function() :
{
}
{
    (
    type() | < VOID >
        )
    < IDENTIFIER > // Function name
        "(" (parameter_declaration() ("," parameter_declaration())*)? ")" "{"
        function_body()
        "}"
    < EOF >
}
```

Déclarer du code

On peut placer du code partout dans une règle. Le code doit être encapsulé dans des accolades.

```
void function() :
{
}
{
    {  System.out.println("Début de fonction"); }
        (
          type() | < VOID > {  System.out.println("Token void rencontré"); }
        )
        //...
}
```

Récupérer un Token

Pour récupérer la valeur d'un Token on crée un objet de type « Token ».

Cette classe est définie lorsque vous compilez votre grammaire.

Elle possède la méthode toString qui permet de récupérer **la valeur** du token (et pas son nom).

Récupérer un Token

```
void function() :
{
  Token t;
}
{
        (
          type() | < VOID >
        )
        t = < IDENTIFIER > // Nom de la fonction
        {
                System.out.println("Nom de la fonction :" +
                t.toString());
        }
        //...
```

Bloc de déclaration des variables utilisées hors des blocs de code

Il n'y a pas d'accolade pour l'assignation

Instruction LOOKAHEAD

L'instruction LOOKAHEAD permet de vérifier un certain nombre de symboles pour prendre une décision en cas d'ambiguïté

LOOKAHEAD(#) : regarde le nombre de symboles indiqué

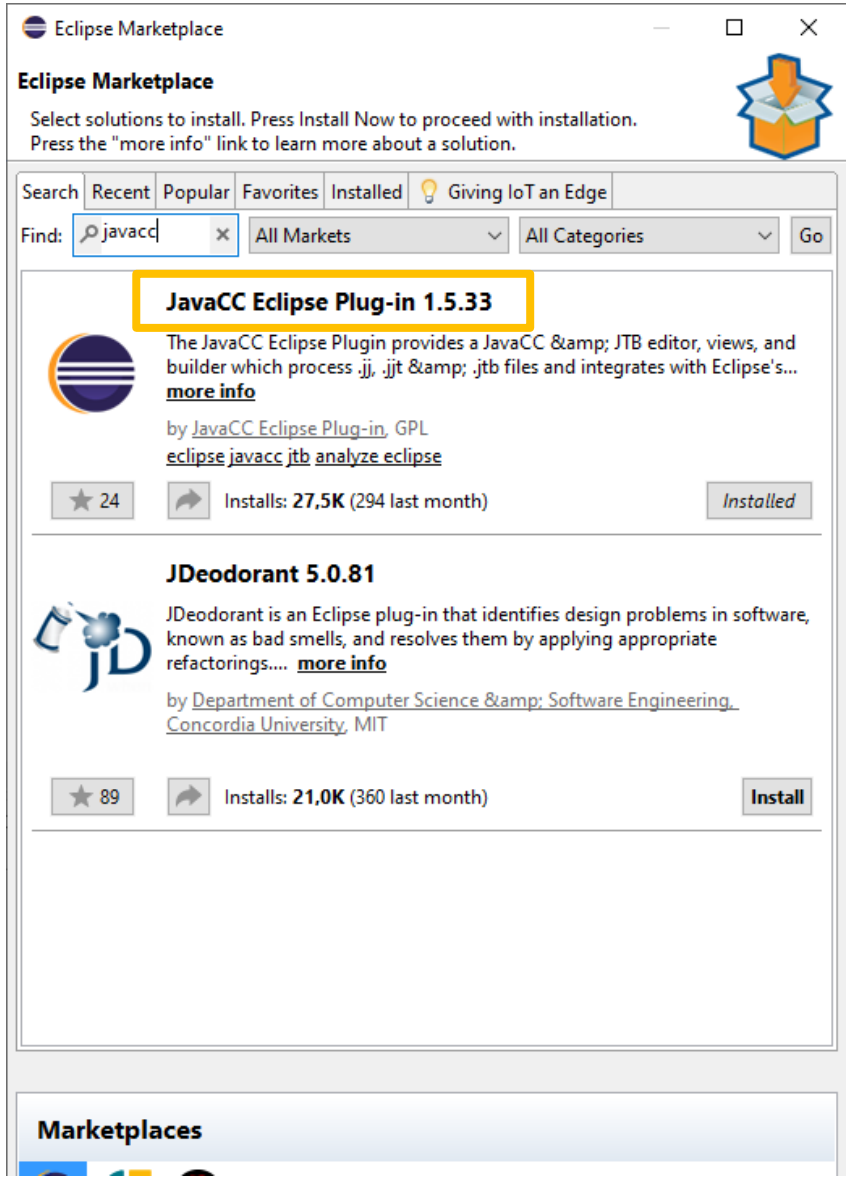
LOOKAHEAD(règle) : tente d'appliquer la règle pour déterminer si l'entrée correspond (plus lourd en ressources)

JavaCC dans Eclipse

Installer JavaCC

Vous devez installer le complément JavaCC dans Eclipse.

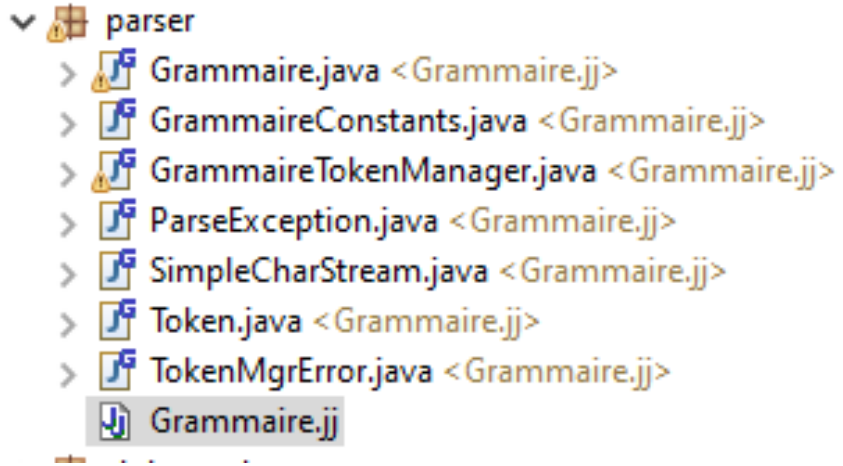
Help > Eclipse Marketplace



Compiler avec JavaCC

Vous devez compiler la grammaire pour pouvoir exécuter votre application.

JavaCC > Compiler avec javacc | jjtree | jtb (Touche F9)



Compiler avec JavaCC

Vous ne devez jamais modifier le code dans les fichiers générés (ceux avec < *.jj > à côté).

Pourquoi ? Parce qu'à la prochaine compilation vos modifications seront effacées.

Exception: Avec la grammaire *java-1.7.jj*, il faut ajouter la classe Token.GTToken dans la classe Token.

Abstract syntaxe tree - AST

Représentation d'un logiciel sous forme d'arbre.

Utilise la syntaxe de la grammaire pour définir les composants de l'arbre.

Toutes les classes de l'arbre implémentent la même interface (ou héritent d'une classe même classe abstraite) ➔ Composite

Parties d'une grammaire Java

<compilation unit> ::= {<package declaration>} {import statements}
<class definitions>

<import statements> ::= import <name> ; {<import statements>}

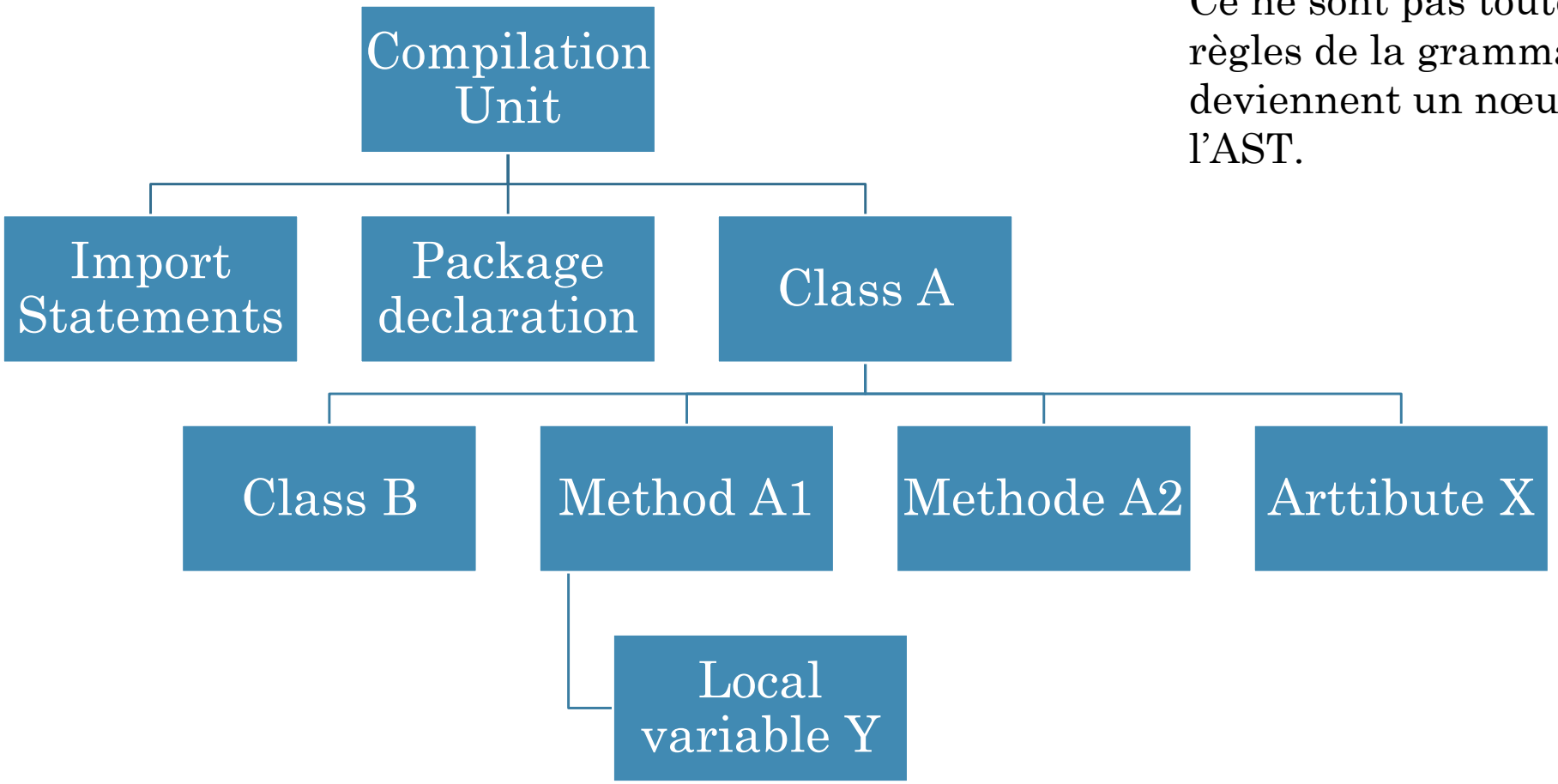
<class definitions> ::= <class definition> {<class definitions>}

<class definition> ::= {<modifiers>} class <name> {extends <name>}
{implements <name list>} "{" {<class member>} "}"

<modifiers> ::= (public | protected | abstract | final | …)
{, <modifiers>}

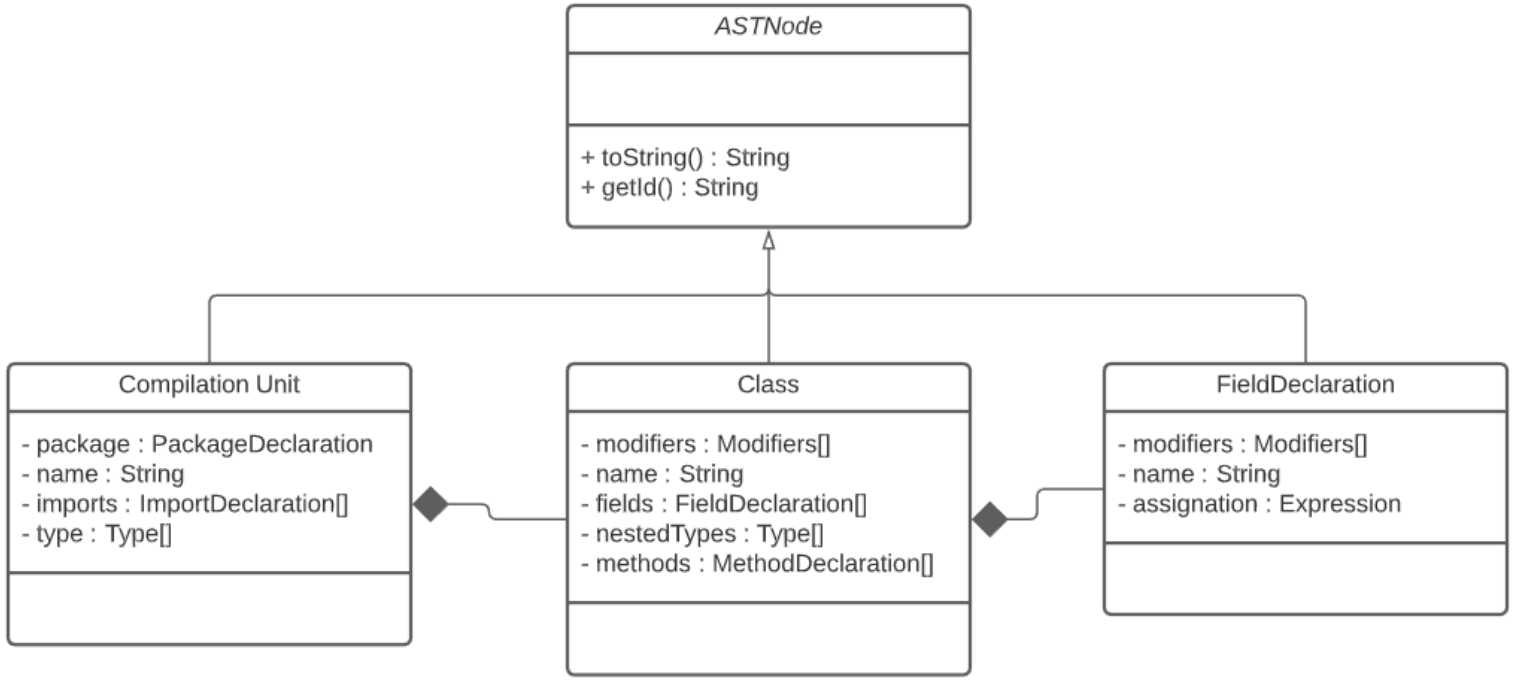
<class member> ::= (<field declaration> | <class definition> |
<method definition> | …) {<class member>}

Parties d'un AST d'un programme Java



Ce ne sont pas toutes les règles de la grammaire qui deviennent un nœud de l'AST.

Diagramme de classe de conception



Construire l'AST avec une pile

La structure récursive de la grammaire peut faire en sorte que l'on s'y perde.

Exemple

```
public class A {
        class B {
                class C {
                        int x;
                }
        }
}
```

Comment garder trace des objets pour que :

- X soit dans C
- C soit dans B
- B soit dans A

Révision pile (stack)

Conteneur de donnée de type LIFO (Last in first out) – dernier arrivé, premier sortie (ex: pile d'assiette)

3 opérations :

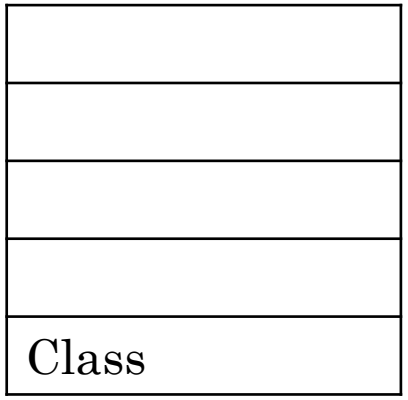
- Empiler (PUSH) : ajoute un objet sur le dessus de la pile
- Retirer (POP) : retire et retourne l'objet sur le dessus de la pile
- Consulter (PEEK) : retourne l'objet sur le dessus de la pile

Utiliser une pile

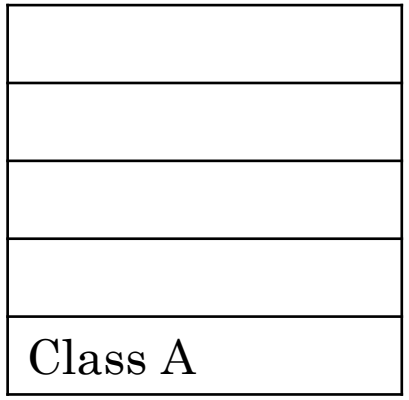
```
(1) public class A (2) {
        (3) int x; (4)
}
```

L'objet actif (sur lequel vous êtes entrain d'extraire de l'information) est le dessus de la pile.

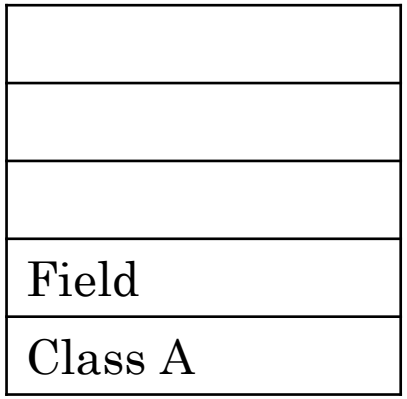
1 : Empiler une classe (push)



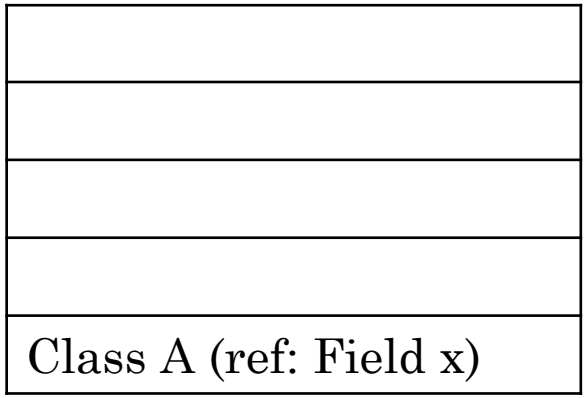
2 : Modifier le nom de A (peek)



3 : Ajouter un attribut (push)



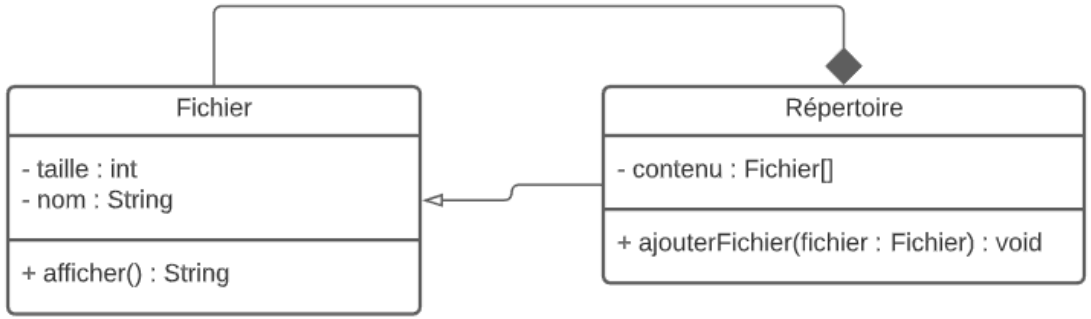
4 : Retirer l'attribut et l'affecter à l'objet en dessous (pop + peek)



Visiteur

Le design pattern Visiteur permet de parcourir une hiérarchie d'objet (comme un arbre) en affectant la responsabilité de l'action à l'extérieur de la classe.

Par exemple, on a une arborescence de fichier et dossier et on veut l'afficher.



Afficher le contenu d'un répertoire

Dans fichier :

```
public void afficher() {
    System.out.println(nom + "[" + taille + " octets]");
}
```

Dans dossier :

```
public void afficher() {
    System.out.println(nom + "[" + taille + " octets]");
    for(Fichier f : contenu)
        f.afficher();
}
```

Impact de cette conception

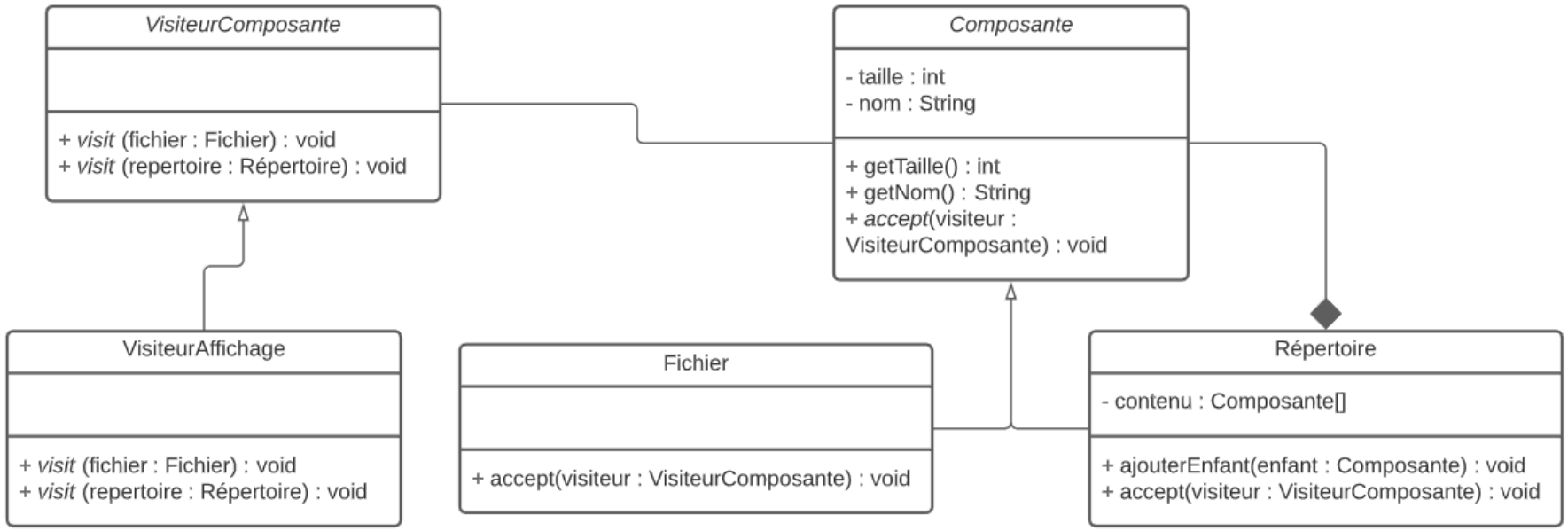
- Si l'on veut gérer l'indentation, il faudra:
  - Ajouter un paramètre
  - Définir dans chaque classe la façon de gérer l'indentation
- Si l'on veut ajouter d'autres opérations lors d'un parcours (exemple : faire une recherche), il faudra :
  - Implémenter une méthode de plus dans chaque classe du composite
  - Répéter la structure de propagation (imaginez un AST avec 15 classes… beaucoup de répétitions)

```
for(Fichier f : contenu)
    f.quelquechose()
```

Patron visiteur

- Le traitement (affichage par exemple) est fait dans une classe à part : le visiteur
- Le visiteur doit effectuer le traitement pour toutes les classes du composite (afficher dossier et afficher fichier)
- Les classes du composite sont responsables de propager le visiteur aux éléments qui les composent

Patron visiteur



Patron visiteur

Dans fichier :

```
public void accept(VisiteurComposante visiteur) {
    visiteur.visit(this);

}
```

Dans dossier :

```
public void accept(VisiteurComposante visiteur){
    visiteur.visit(this);
    for(Composante c : contenu)
         c.accept(visiteur);

}
```

Patron visiteur

Dans visiteur:

```
public void visit(Fichier fichier) {
    System.out.println("Fichier : " + fichier.getNom() + " [" +
    fichier.getTaille() + " octets]");
}

public void visit(Repertoire repertoire) {
    System.out.println("Répertoire : " + repertoire.getNom());
}
```

Impact de cette conception

- Ajouter de nouvelles opérations : il suffit de créer un nouveau visiteur qui hérite de VisiteurComposante.

- La logique de propagation est implémentée une seule fois par classe plutôt que d'être répétée à chaque traitement.

- Très utile aussi pour proposer une API comme les clients pourront traverser la hiérarchie avec un traitement personnalisé (ex. rechercher un fichier avec un nom précis).

Aller plus loin avec le visiteur

On peut aussi définir les méthodes

- `previsit(Composante composante)`
- `postvisite(Composante composante)`

Elles s'exécutent respectivement avant et après la méthode visit (généralement la propagation est inclue entre pre et post visit).

Par exemple, on peut facilement gérer les niveaux d'indentation d'un affichage avec ces méthodes.

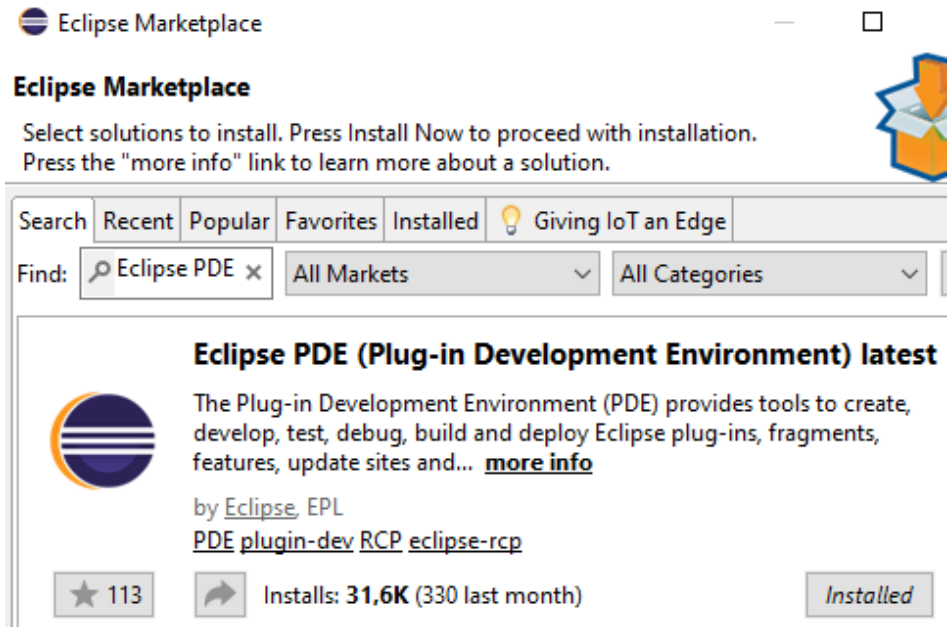
AST (Eclipse)

AST (Eclipse)

- Eclipse contient une API permettant de manipuler des arbres syntaxiques d'un programme en Java.
  - Permet d'utiliser le patron visiteur.
- Avec les outils de développement de plugin, on peut écrire un programme utilisant les AST sur un projet Eclipse.

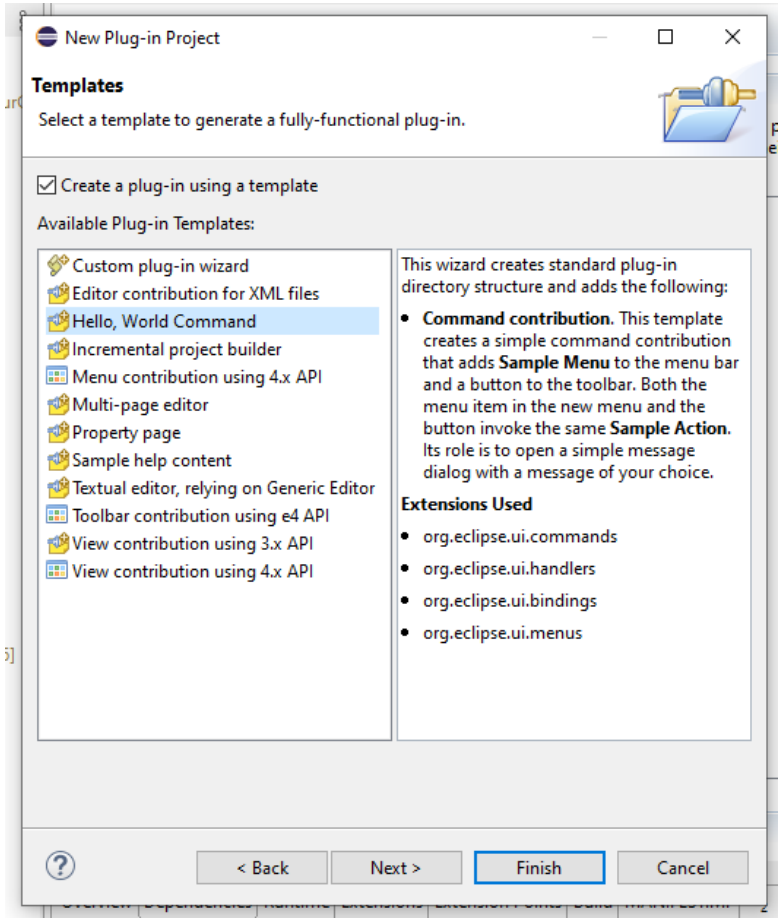
Installer Eclipse PDE

- **Help > Eclipse Marketplace…**
- Chercher *Eclipse PDE*
- Installer



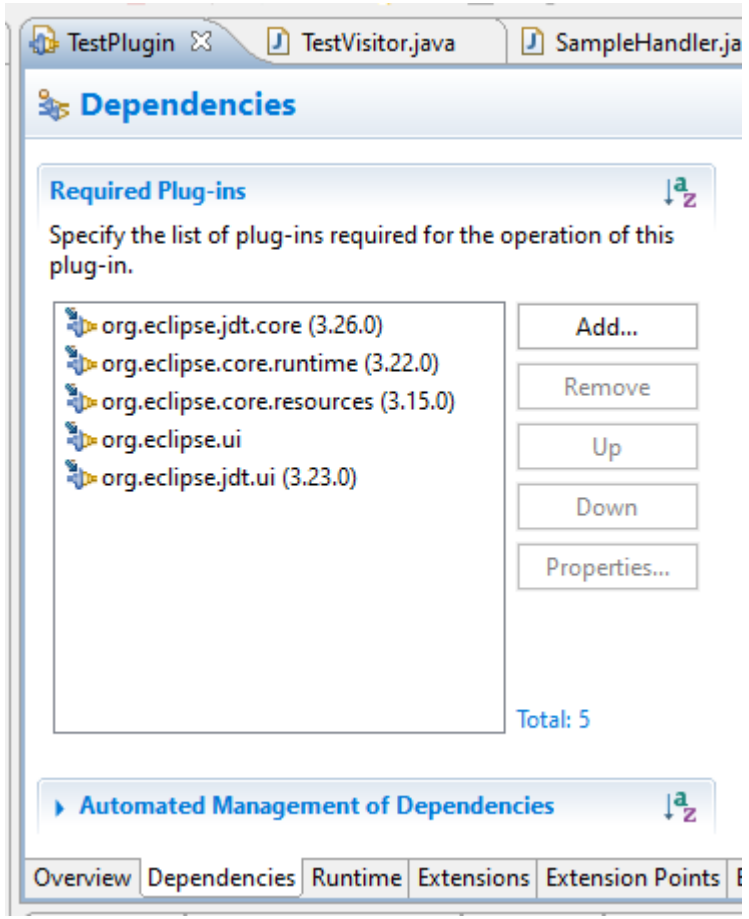
Créer un projet

- **File > New > Plugin Project**
- Nommer le projet et cliquer *Next* jusqu'à la page **Template**.
- Recommandé : sélectionner le template *Hello, World Command*
- **Finish**



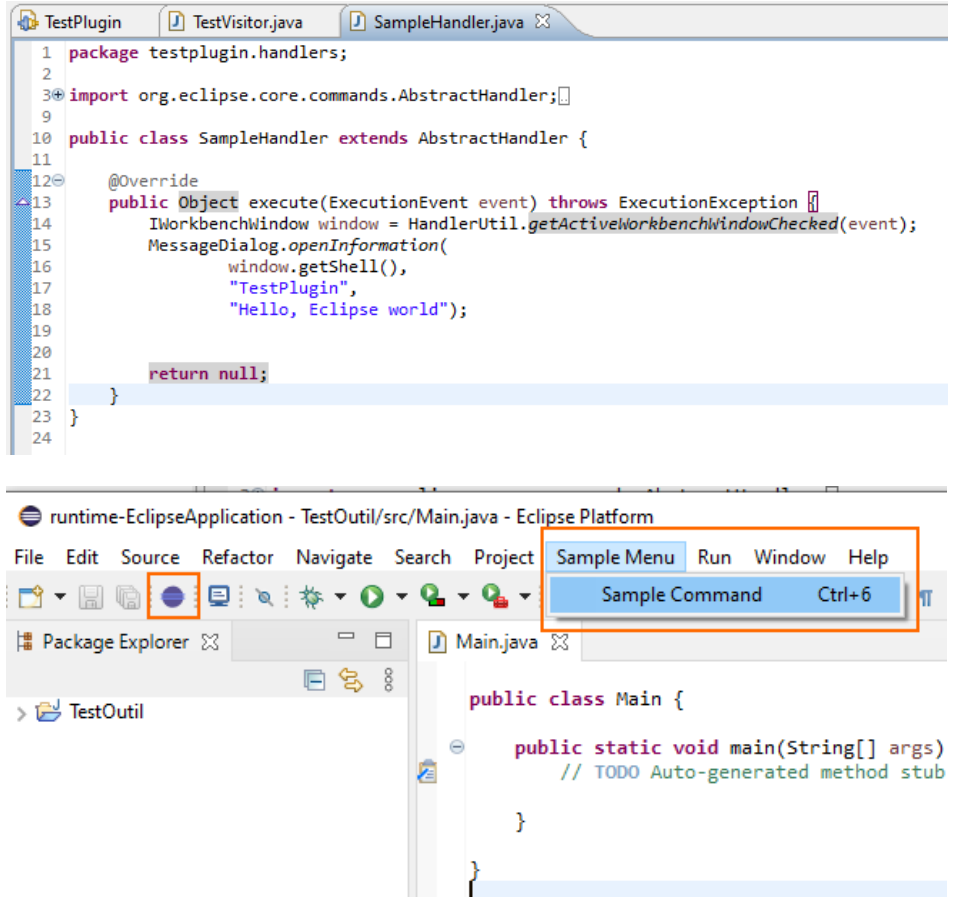
Ajouter les dépendences

- Ajouter ces dépendances dans le fichier **plugin.xml**



SampleHandler

- Lorsque le projet est exécuté, une deuxième instance d'Eclipse est lancée.
- Cette instance contient la commande SampleCommand.
- Lorsque cette commande est actionnée, la méthode *SampleHandler.execute(event)* est appelée.
- Vous pouvez modifier le code de la méthode execute() pour effectuer le traitement souhaité.



Visiteur

- Vous pouvez créer une classe qui étend la classe *org.eclipse.jdt.core.dom.ASTVisitor*.
- Vous pouvez redéfinir les méthodes visit correspondant aux composantes que vous voulez analyser.
- Vous pourrez instancier votre visiteur dans la classe SampleHandler et l'appliquer aux unités de compilations.

```
 6  import org.eclipse.jdt.core.dom.ASTVisitor;
 7  import org.eclipse.jdt.core.dom.AnonymousClassDeclaration;
 8  import org.eclipse.jdt.core.dom.FieldDeclaration;
 9  import org.eclipse.jdt.core.dom.IBinding;
10  import org.eclipse.jdt.core.dom.IMethodBinding;
11  import org.eclipse.jdt.core.dom.ITypeBinding;
12  import org.eclipse.jdt.core.dom.IVariableBinding;
13  import org.eclipse.jdt.core.dom.MethodDeclaration;
14  import org.eclipse.jdt.core.dom.MethodInvocation;
15  import org.eclipse.jdt.core.dom.SimpleName;
16  import org.eclipse.jdt.core.dom.SingleVariableDeclaration;
17  import org.eclipse.jdt.core.dom.TypeDeclaration;
18  import org.eclipse.jdt.core.dom.VariableDeclarationFragment;
19
20
21
22
23
24
25  public class StructuralVisitor extends ASTVisitor {
26
27
28
30      public boolean visit(TypeDeclaration node) {
61
63      public void endVisit(TypeDeclaration node) {
71
73      public boolean visit(AnonymousClassDeclaration node) {
77
79      public boolean visit(MethodDeclaration node) {
94
96      public void endVisit(MethodDeclaration node) {
102
104     public boolean visit(MethodInvocation node) {
141
143     public boolean visit(SimpleName node) {
```

Accéder aux unités de compilation

- IProject[] org.eclipse.core.resources.ResourcesPlugin.getWorkspace() .getRoot().getProjects()
  - Accéder aux projets Eclipse
- boolean org.eclipse.core.resources.IProject.isNatureEnabled("org.eclipse.jdt.core.javanature")
  - Vérifie si un projet est un projet Java
- IPackageFragment[] org.eclipse.jdt.core.IJavaProject.getPackageFragments()
- ICompilationUnit[] org.eclipse.jdt.core.IPackageFragment.getCompilationUnits()
- CompilationUnit ast.handlers.SampleHandler.parse(ICompilationUnit unit)
- org.eclipse.jdt.core.dom.ASTNode.accept(ASTVisitor visitor)
  - CompilationUnit est un ASTNode et peut accepter votre visiteur.

ASTView

Plugin optionnel

ASTView

- Permet de visualiser et naviguer dans les AST.
  - Fort utile pour se familiariser avec la structure des AST d'Eclipse.
- Pour l'installer
  - Aller sur cette page.
  - Copier le lien sous Update Site.
  - Dans Eclipse, aller dans **Help > Install New Software… > Add…**
  - Coller le lien dans le champ **Location**. Puis cliquer sur **Add**.
  - Cocher ASTView et cliquer Next. Poursuivez jusqu'à la fin de l'installation.
- Ouvrir la vue ASTView
  - Dans Eclipse, aller dans **Window > Show View > Other… > ASTView**