NFA035
Eclipse

S. Rosmorduc

Programme

- Initiation à l'environnement Eclipse
- tests et JUnit
- la programmation objet
- les collections en java (ou comment regrouper des données)
- les entrées/sorties en java (ou comment lire et écrire dans des fichiers)
- interfaces graphiques en java (Swing)

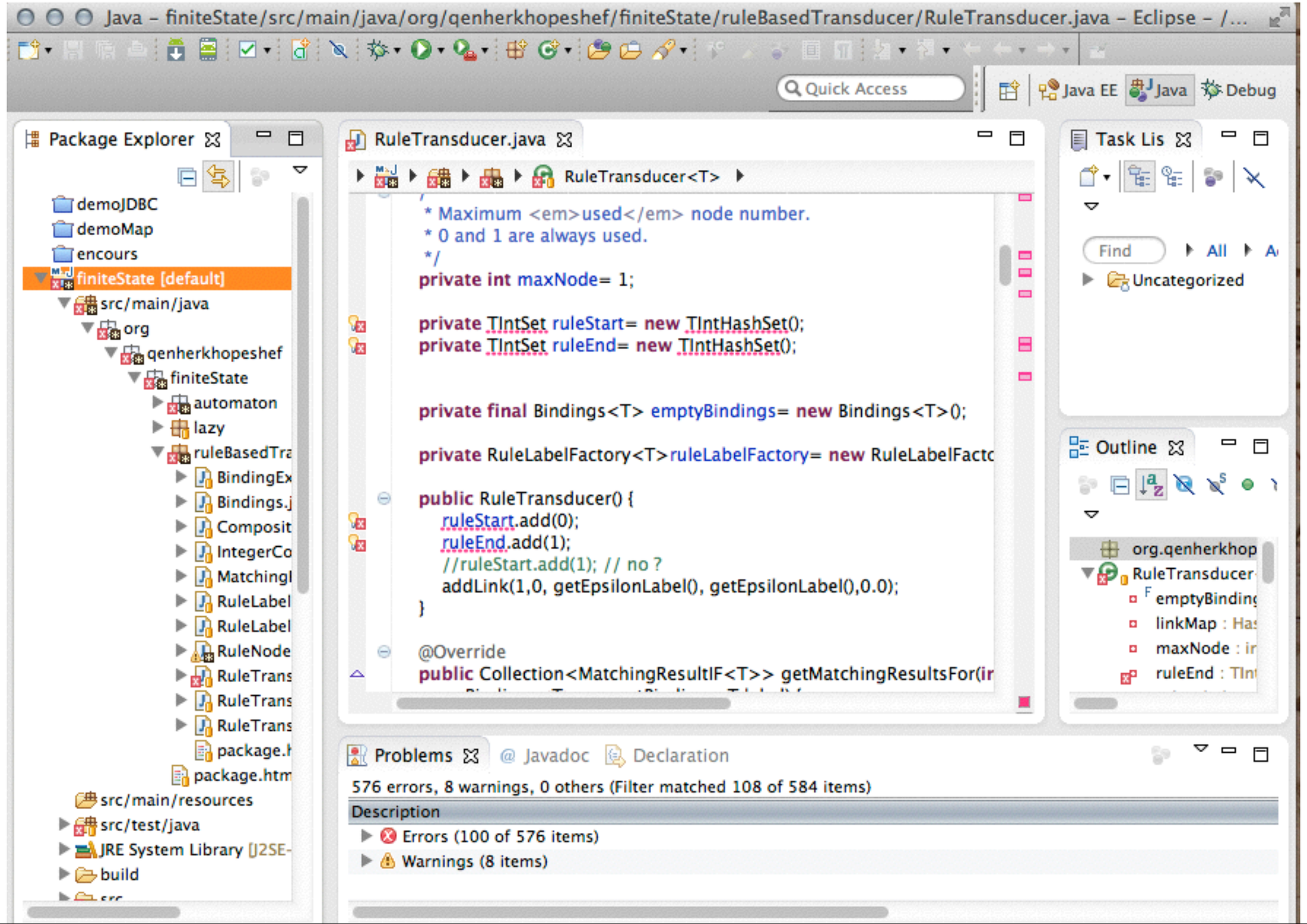
Aujourd’hui

- Mise en train:
  - l’outil que nous utiliserons : eclipse
  - un instrument précieux pour le développeur : le débuggeur
  - un peu de java quand même : les packages

Initiation à eclipse

- IDE : « environnement de développement intégré »
- concurrents : netbeans, intellij
- permet d'effectuer de manière intégrée la plupart des opérations liées à la production de logiciel
  - écriture, compilation
  - test
  - publication…

Eclipse

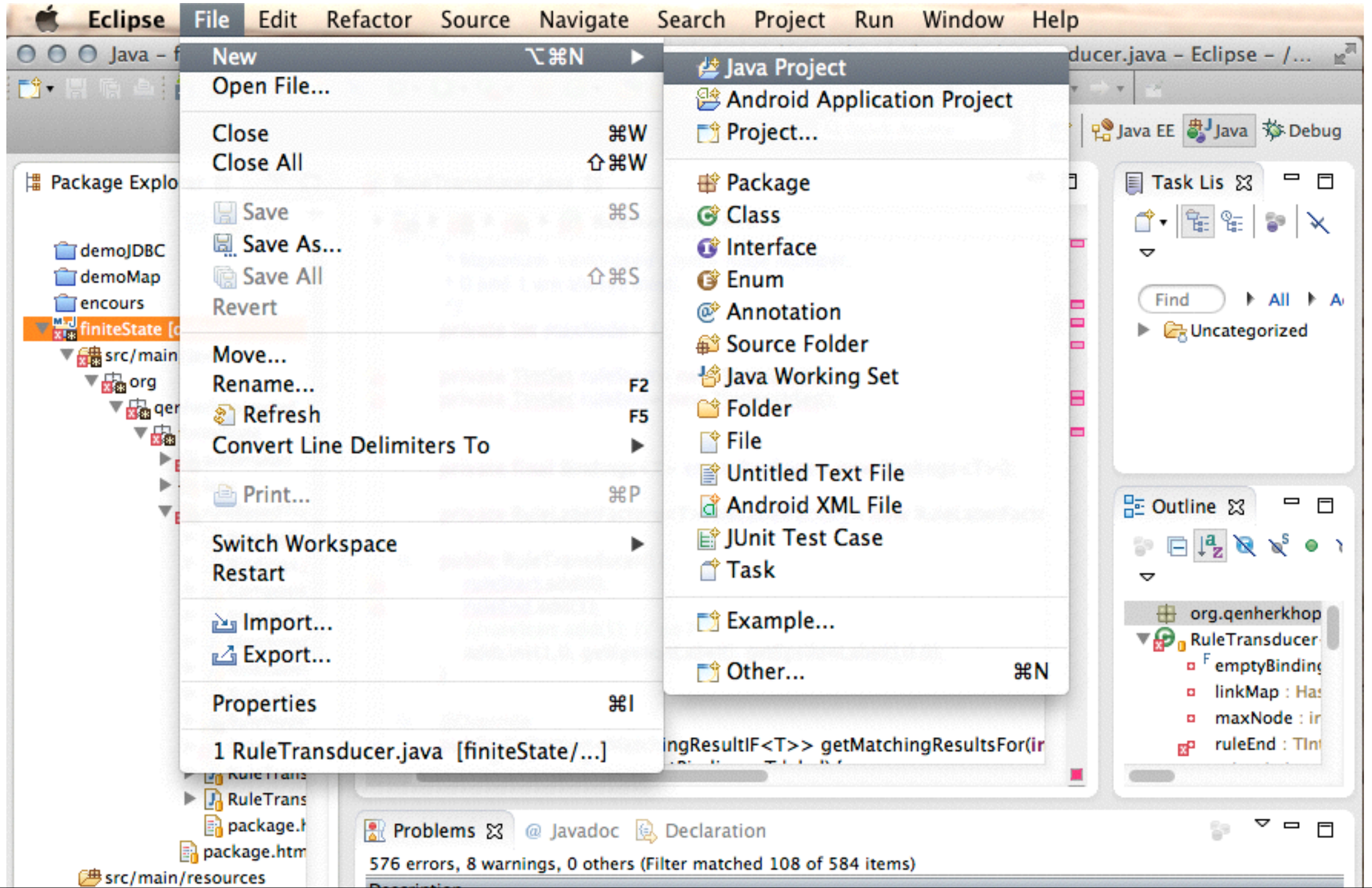

Java – finiteState/src/main/java/org/qenherkhopeshef/finiteState/ruleBasedTransducer/RuleTransducer.java – Eclipse – /...
Quick Access
Java EE
Java
Debug
Package Explorer
demoJDBC
demoMap
encours
finiteState [default]
src/main/java
org
qenherkhopeshef
finiteState
automaton
lazy
ruleBasedTra
BindingEx
Bindings.j
Composit
IntegerCo
MatchingI
RuleLabel
RuleLabel
RuleNode
RuleTrans
RuleTrans
RuleTrans
package.h
package.htm
src/main/resources
src/test/java
JRE System Library [J2SE-
build
RuleTransducer.java
RuleTransducer<T>
* Maximum used node number.
* 0 and 1 are always used.
*/
private int maxNode= 1;
private TIntSet ruleStart= new TIntHashSet();
private TIntSet ruleEnd= new TIntHashSet();
private final Bindings<T> emptyBindings= new Bindings<T>();
private RuleLabelFactory<T>ruleLabelFactory= new RuleLabelFactc
public RuleTransducer() {
ruleStart.add(0);
ruleEnd.add(1);
//ruleStart.add(1); // no ?
addLink(1,0, getEpsilonLabel(), getEpsilonLabel(),0.0);
}
@Override
public Collection<MatchingResultIF<T>> getMatchingResultsFor(ir
Task Lis
Find
All
Uncategorized
Outline
org.qenherkhop
RuleTransducer
emptyBinding
linkMap : Has
maxNode : ir
ruleEnd : TInt
Problems
Javadoc
Declaration
576 errors, 8 warnings, 0 others (Filter matched 108 of 584 items)
Description
Errors (100 of 576 items)
Warnings (8 items)

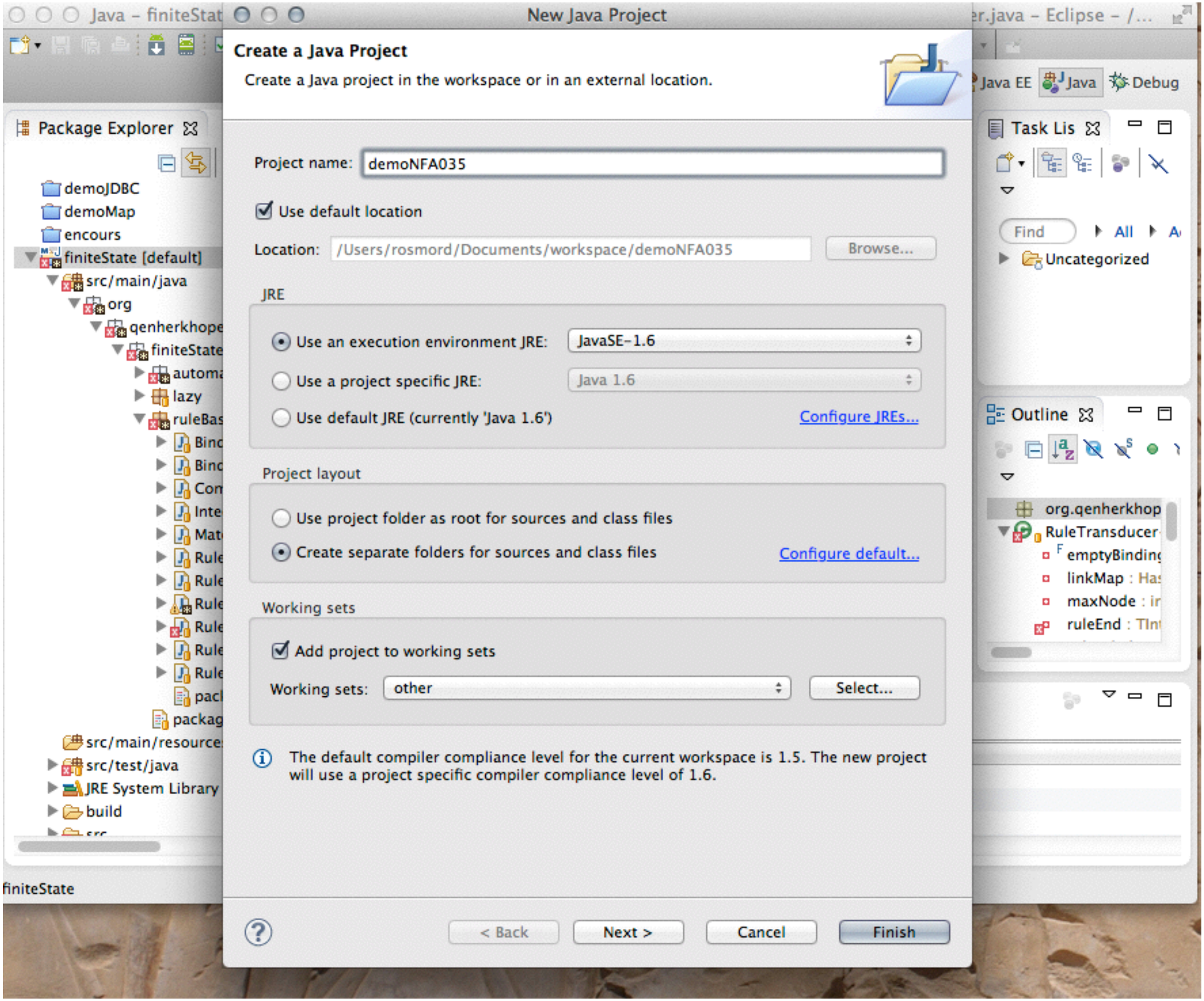
Notion de projet

- Dans eclipse, les fichiers sont créés dans des **projet**

- un « vrai » programme java comporte généralement beaucoup de fichiers « .java » : on les regroupe en projets

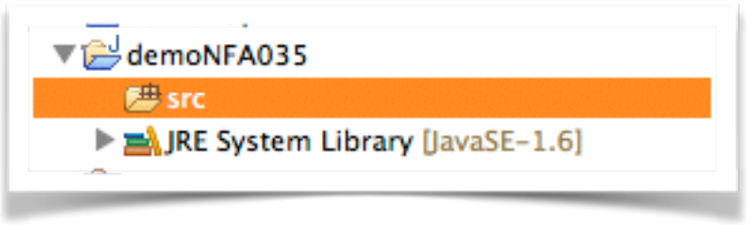
Création d'un projet




New Java Project
Create a Java Project
Create a Java project in the workspace or in an external location.
Project name: demoNFA035
Use default location
Location: /Users/rosmord/Documents/workspace/demoNFA035
Browse...
JRE
Use an execution environment JRE: JavaSE-1.6
Use a project specific JRE: Java 1.6
Use default JRE (currently 'Java 1.6')
Configure JREs...
Project layout
Use project folder as root for sources and class files
Create separate folders for sources and class files
Configure default...
Working sets
Add project to working sets
Working sets: other
Select...
The default compiler compliance level for the current workspace is 1.5. The new project will use a project specific compiler compliance level of 1.6.
< Back
Next >
Cancel
Finish
Package Explorer
demoJDBC
demoMap
encours
finiteState [default]
src/main/java
org
src/main/resources
src/test/java
JRE System Library
build
Java EE
Java
Debug
Task Lis
Find
All
Uncategorized
Outline
RuleTransducer
emptyBinding
linkMap : Has
maxNode : ir
ruleEnd : TInt

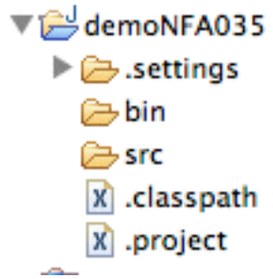
Structure d'un projet

- contient des fichiers cachés de configuration (.project, .classpath)



vue logique : « package explorer »

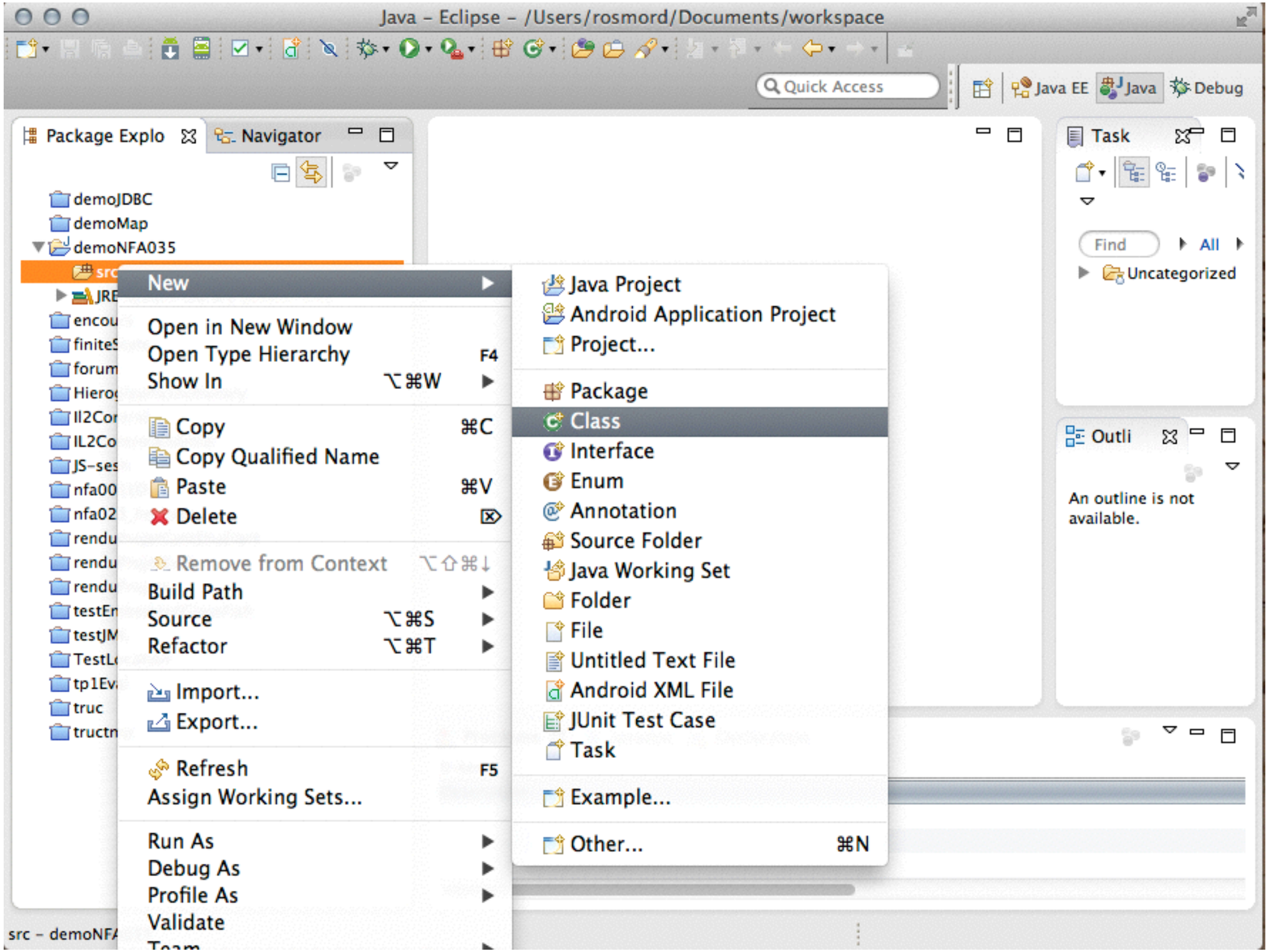
- contient un dossier src où on place les sources java

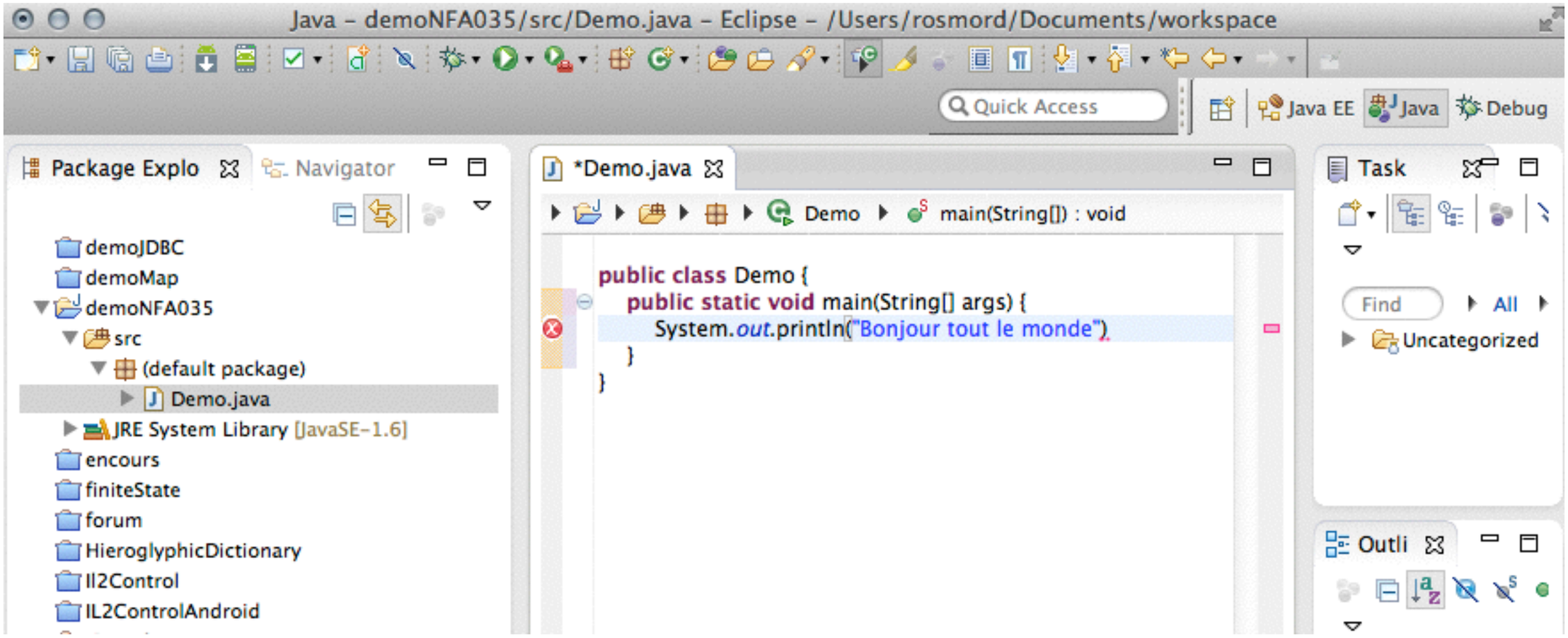


vue réelle (« Navigator »)

- les « .class » sont automatiquement stockés dans le dossier « bin »

Créer une classe





- le code est compilé « à la volée »
- les erreurs de syntaxe sont signalées en « temps réel ».

Lancer le programme

- Sélectionner le fichier qui contient le « main »
- clic droit, plus « Run as/Java Application »

Le debugger

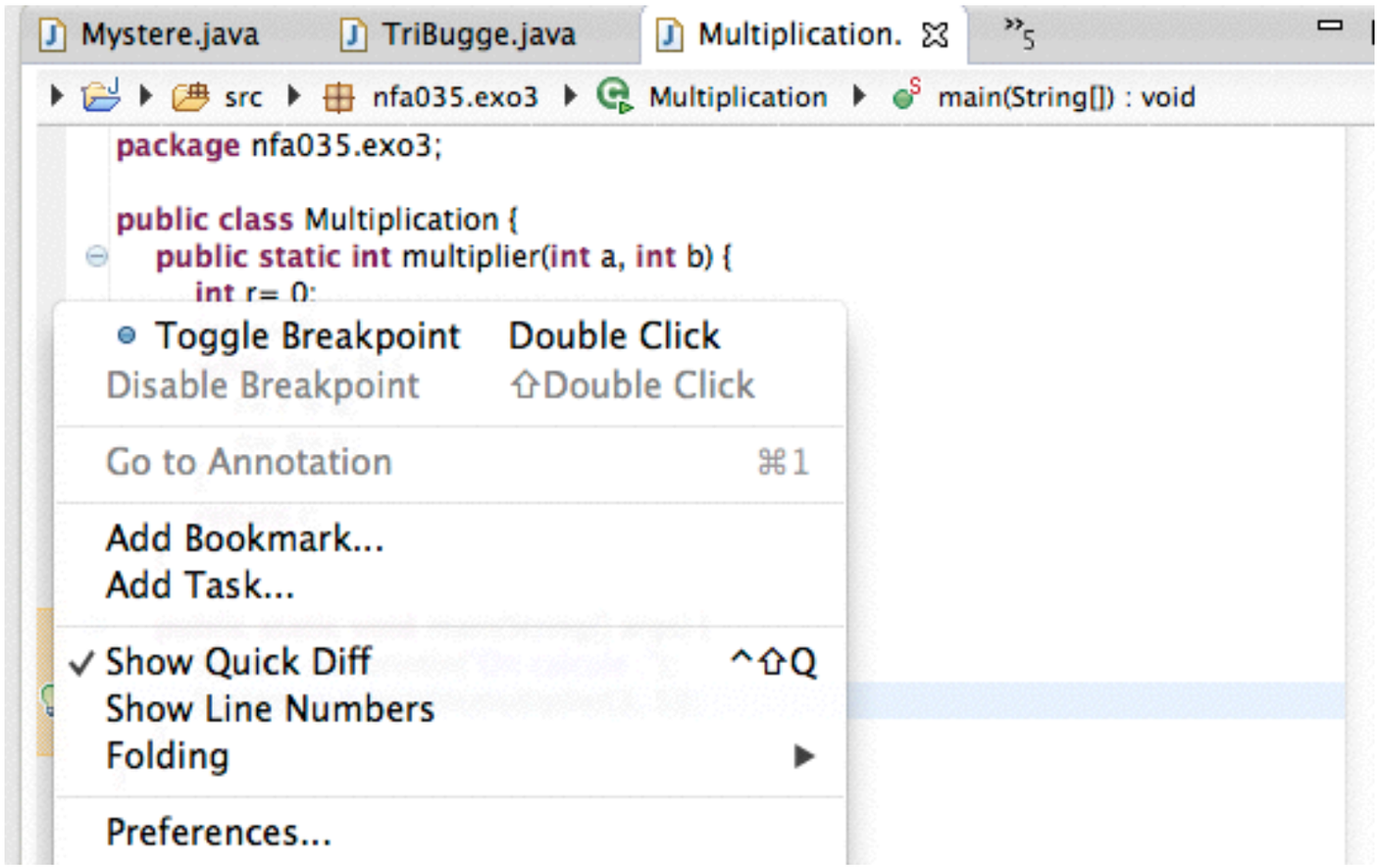
- permet de visualiser la valeur des variables au cours de l'exécution du programme

- principe de base

  - on pose des « points d'arrêt » (breakpoints)

  - quand le programme atteint un point d'arrêt, il est suspendu

  - on peut alors visualiser les variables, exécuter le programme en mode « pas à pas », etc…


Mystere.java
TriBugge.java
Multiplication.
src
nfa035.exo3
Multiplication
main(String[]) : void
package nfa035.exo3;
public class Multiplication {
public static int multiplier(int a, int b) {
int r= 0;
Toggle Breakpoint Double Click
Disable Breakpoint ⇧Double Click
Go to Annotation ⌘1
Add Bookmark...
Add Task...
Show Quick Diff ^⇧Q
Show Line Numbers
Folding
Preferences...

Pile des appels

Valeurs des variables

Debug – nfa035_tp1/src/nfa035/exo3/Multiplication.java – Eclipse – /Users/rosmord/Docume

Quick Access | Java EE | Java | Debug

Debug | Servers

Multiplication [Java Application]
nfa035.exo3.Multiplication at localhost:62152
Thread [main] (Suspended (breakpoint at line 5
Multiplication.multiplier(int, int) line: 5
Multiplication.main(String[]) line: 16
/System/Library/Java/JavaVirtualMachines/1.6.0.jc

Variables | Breakpoints

Name	Value
a	3
b	5

Mystere.java | TriBugge.java | Multiplication. | »5

src ▸ nfa035.exo3 ▸ Multiplication ▸ multiplier(int, int) : int

```
package nfa035.exo3;

public class Multiplication {
    public static int multiplier(int a, int b) {
        int r= 0;
        int n= 0;
        while (n < b) {
            r= r + a;
            n= n++;
        }
        return r;
    }

    public static void main(String[] args) {
        System.out.println("On calcule :");
```

Outline

nfa035.exo3
Multiplication
main(String[]) : voi
multiplier(int, int)

ligne actuelle

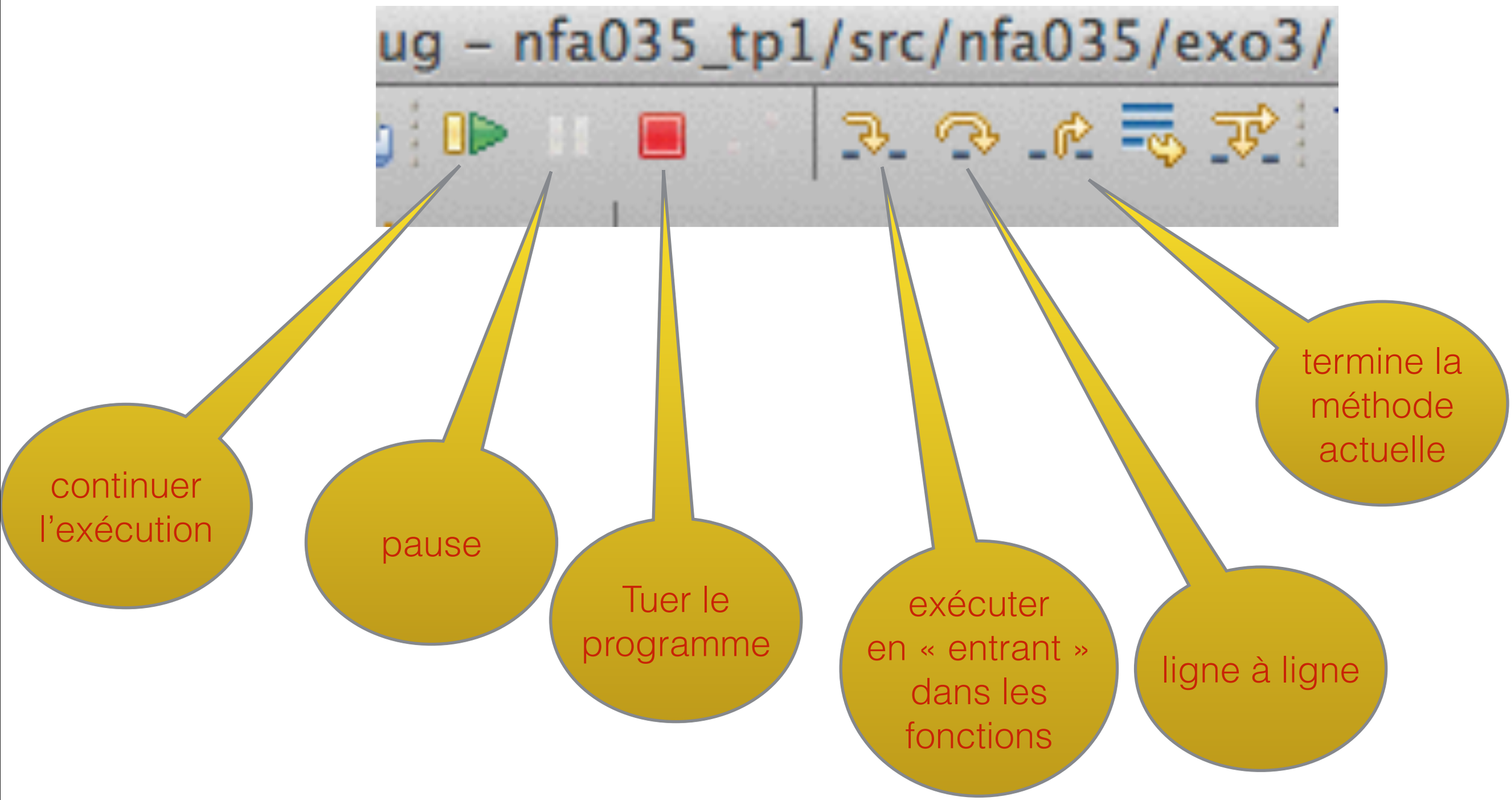
code

Console | JUnit | Search | Call Hierarchy

Multiplication [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (16 févr. 2014 19

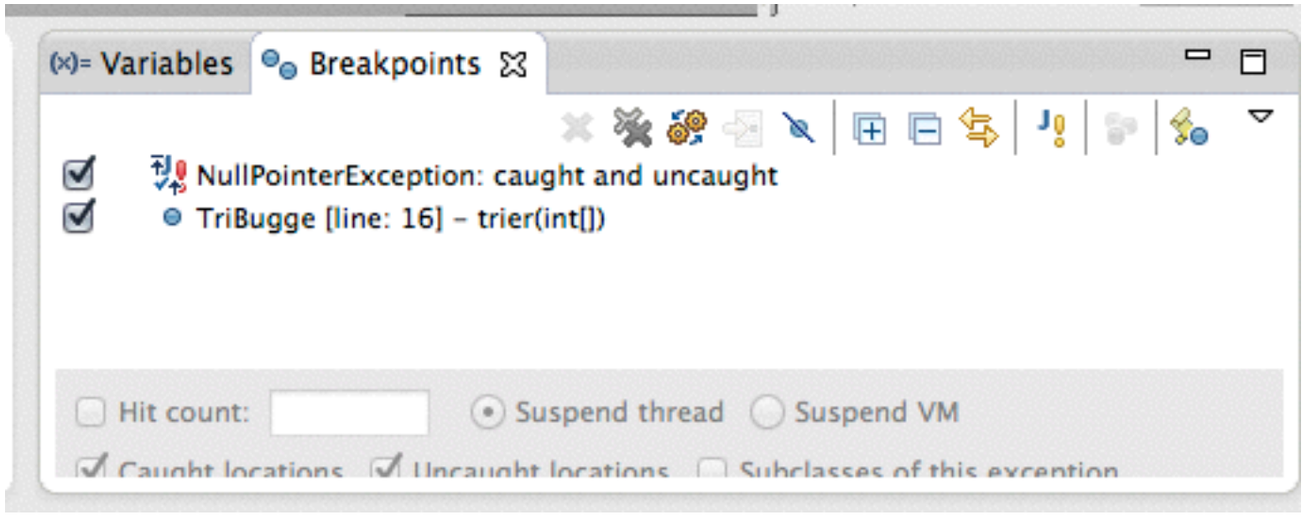
On calcule :

Débugger : barre de commande

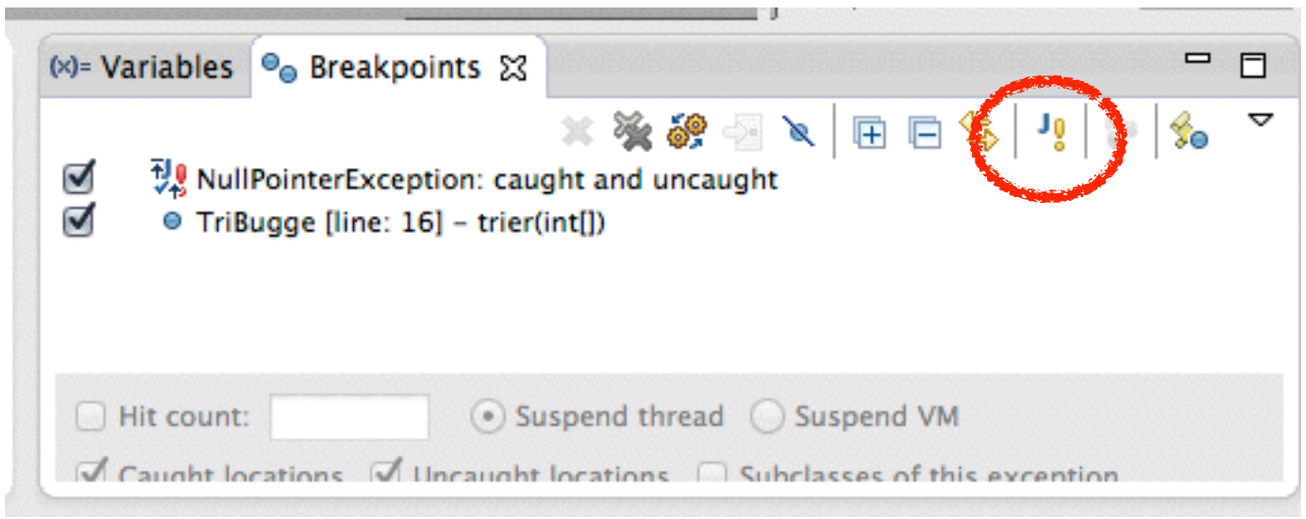


Debugger : gestion des points d'arrêt

- On peut supprimer ou désactiver les points d'arrêt



- On peut poser des points d'arrêt sur des exceptions (très utile!)



Les packages

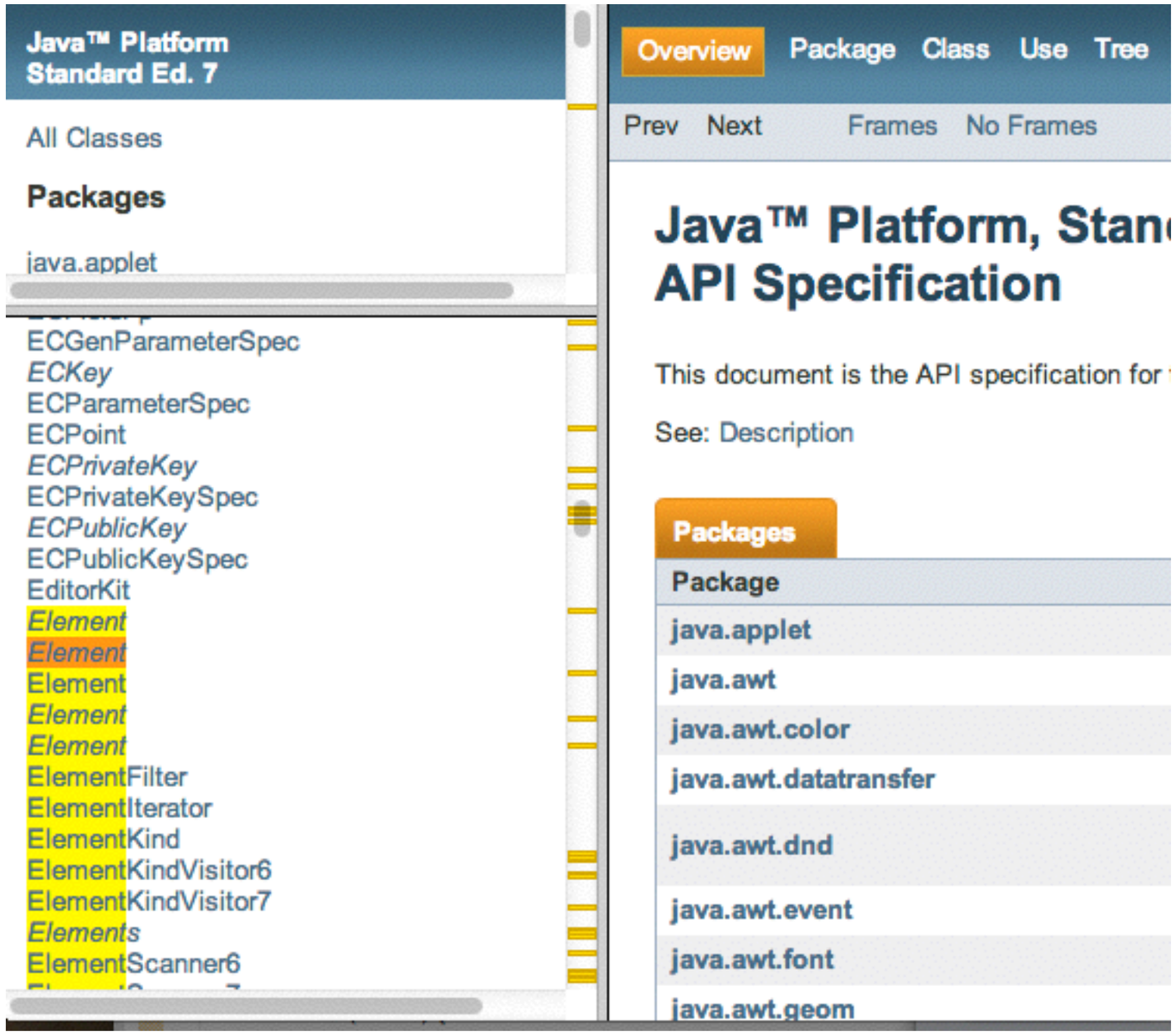
- Problème : un « vrai » programme utilise beaucoup de classes

- il utilise aussi souvent plusieurs bibliothèques téléchargées sur le web, qui contiennent elles-même des classes

- risque (certitude !!) de *collisions* dans les noms : on aura plusieurs classes avec le même nom!

Exemple

- Cinq classes qui s'appellent « Element »
- Deux qui s'appelent List…
- Comment les distinguer ?



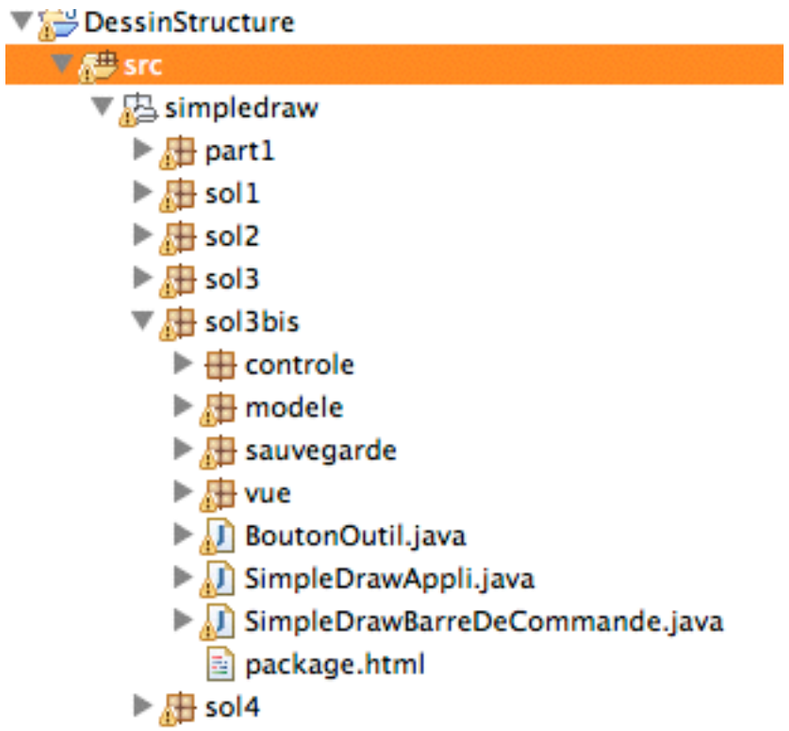
Solution : les package

- On regroupe les classes de manière thématique dans des *packages*
- par convention, le nom d'un package commence par une **minuscule**
- C'est très proche des *dossiers* que vous utilisez pour regrouper les fichiers

- Un package peut contenir des classes ou d'autres packages
- Exemples:
  - java.text : package contenant les classes pour manipuler du texte
  - javax.swing : package de base pour les classes d'interface utilisateur
  - javax.swing.border : package contenant toutes les classes représentant des « bords » de fenêtre

Exemple

- Logiciel de dessin:
  - un package pour le « modele » (la représentation du dessin en mémoire)
  - un package pour la « vue » (son affichage)
  - un package pour le code de sauvegarde du dessin sur fichier



Noms « réels » des classes

- le nom complet d'une classe est le nom de la classe, précédé de celui du package qui la contient
- un sous package a comme nom le nom de son parent, suivi d'un « . », suivi du nom du package

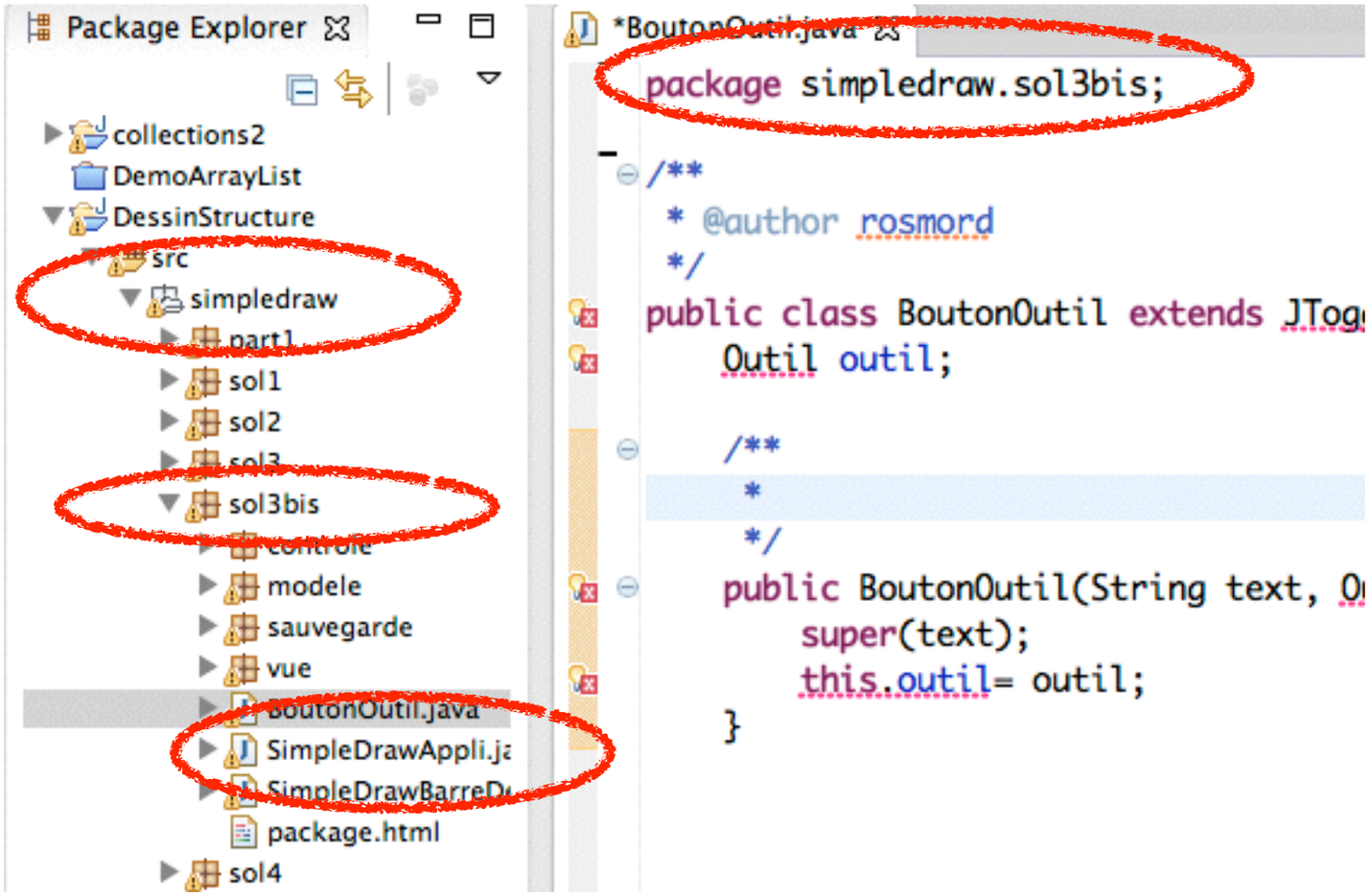
- Exemples
  - classe **java.awt.List** : représente une liste dans une interface graphique awt.
    - package java.awt, dans le package « java » (pour les bibliothèques standards)
  - « classe » **java.util.List** : liste d'éléments en mémoire

Modes d’organisation

- par thème : dans une application de gestion de notes, un package pour la gestion des étudiants, un autre pour ce qui concerne les matières…

- par couche : un package pour l’interface utilisateur, un package pour la logique du programme, un package pour l’accès aux données

Création de packages



- On crée un dossier par package en respectant la hiérarchie
- Dans chaque classe, on déclare son package
- Chic, eclipse le fait tout seul !

Utilisation d’une classe dans un autre package que le sien

- On peut toujours donner à une classe son nom complet:

  public static int somme(**java.util.List** maListe) {…}

- mais c’est pénible

- autre solution : import.

import

- entre la ligne qui déclare le package et le début de la classe, on peut *importer* des classes

- c'est purement syntaxique: ici signifie que dans la classe BoutonOutil, « Outil » désigne la classe simpledraw.sol3bis.controle.Outil

```
package simpledraw.sol3bis;

import javax.swing.JToggleButton;

import simpledraw.sol3bis.controle.Outil;


/**
 * @author rosmord
 */
public class BoutonOutil extends JToggleButton {
    Outil outil;
```

import

- On peut utiliser le caractère « * » pour importer toutes les classes d'un package (mais pas celles des sous packages)

- ex: import java.util.*; // import List, Set….

Le package par défaut

- Une classe qui n'a pas explicitement de package est dans le package par défaut.
- Il n'a pas de nom, le pauvre
- C'est celui que vous avez utilisé jusqu'à présent…
- mais c'est fini: on ne peut pas importer une classe qui est dans le package par défaut
- du coup, il faut éviter de l'utiliser

public, private et rien

- public : la classe et la méthode est visible par tout le monde

- private : une méthode ou un champ private n'est visible que depuis la classe où il est défini

- protected (on en parle plus tard)

- (rien) : quand une méthode n'est ni publique ni private, elle est « publique dans son package, private ailleurs »