

NFA032

Eclipse

V. Aponte d'après S. Rosmorduc

Aujourd'hui

- Mise en train:
 - l'outil que nous utiliserons : eclipse
 - un peu de java quand même : les packages

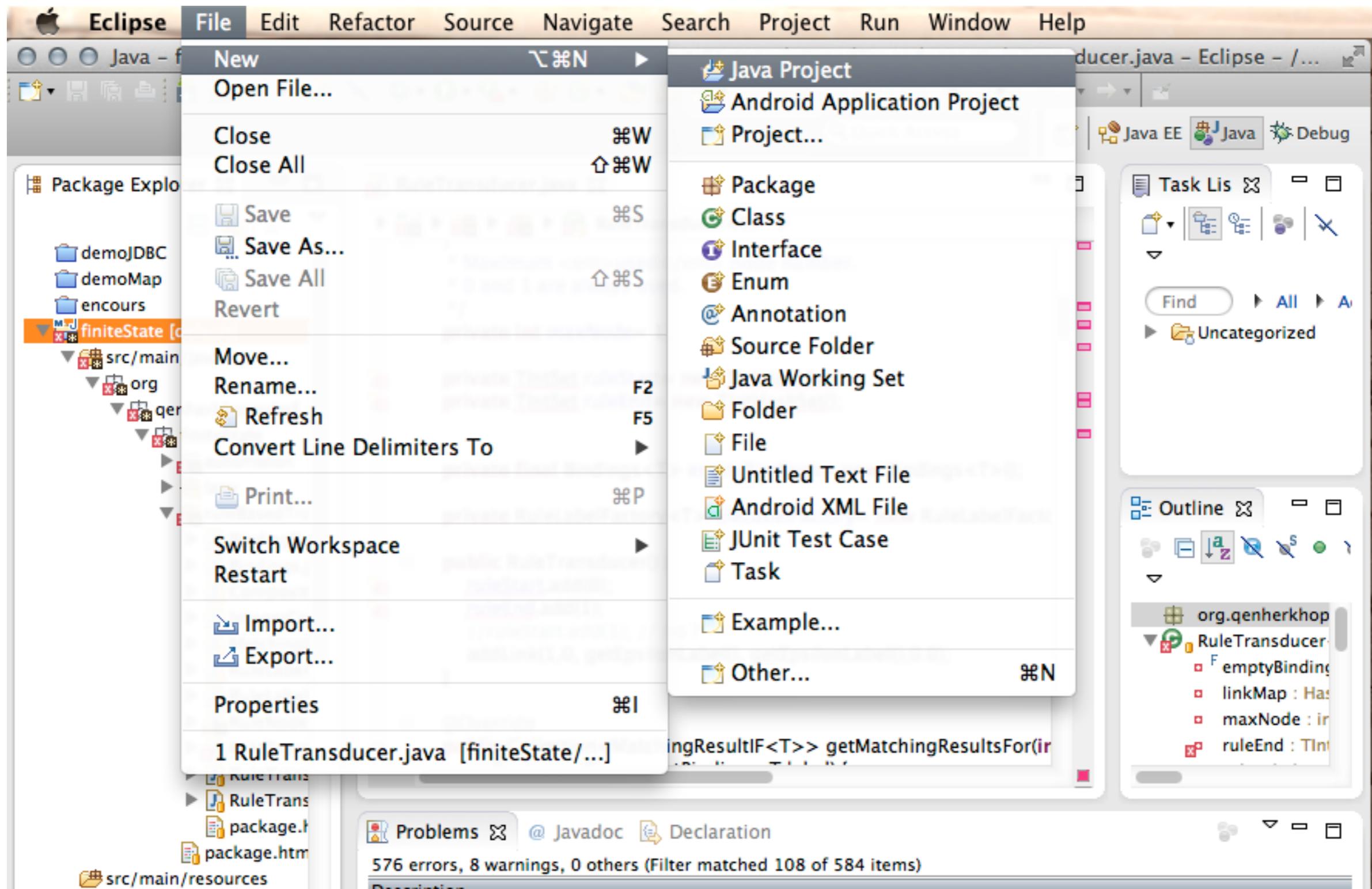
Initiation à Eclipse

- 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...

Notion de projet

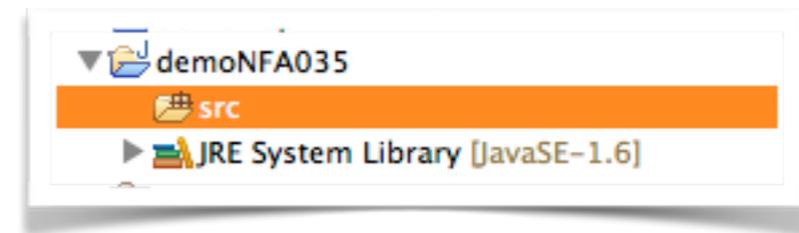
- Dans eclipse, les fichiers sont créés et regroupés dans des **projet**
- un « vrai » programme java comporte généralement beaucoup de fichiers « .java » : on les regroupe en projets

Création d'un projet

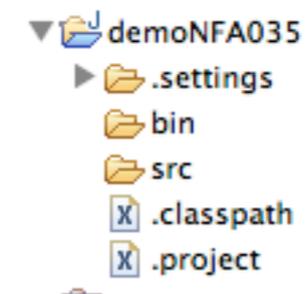


Structure d'un projet

- contient des fichiers cachés de configuration (.project, .classpath)
- contient un dossier src où on place les sources java
- les « .class » sont automatiquement stockés dans le dossier « bin »

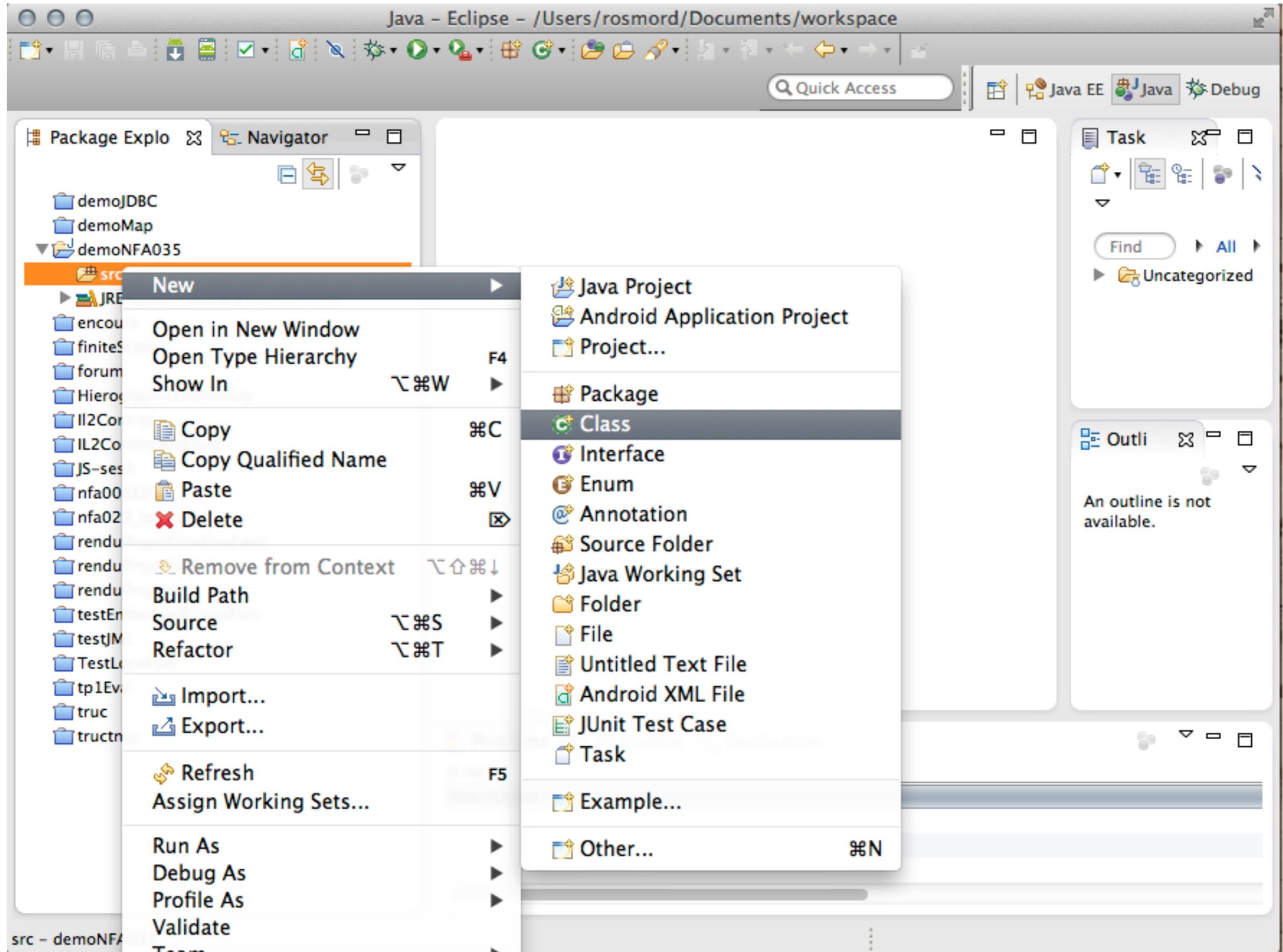


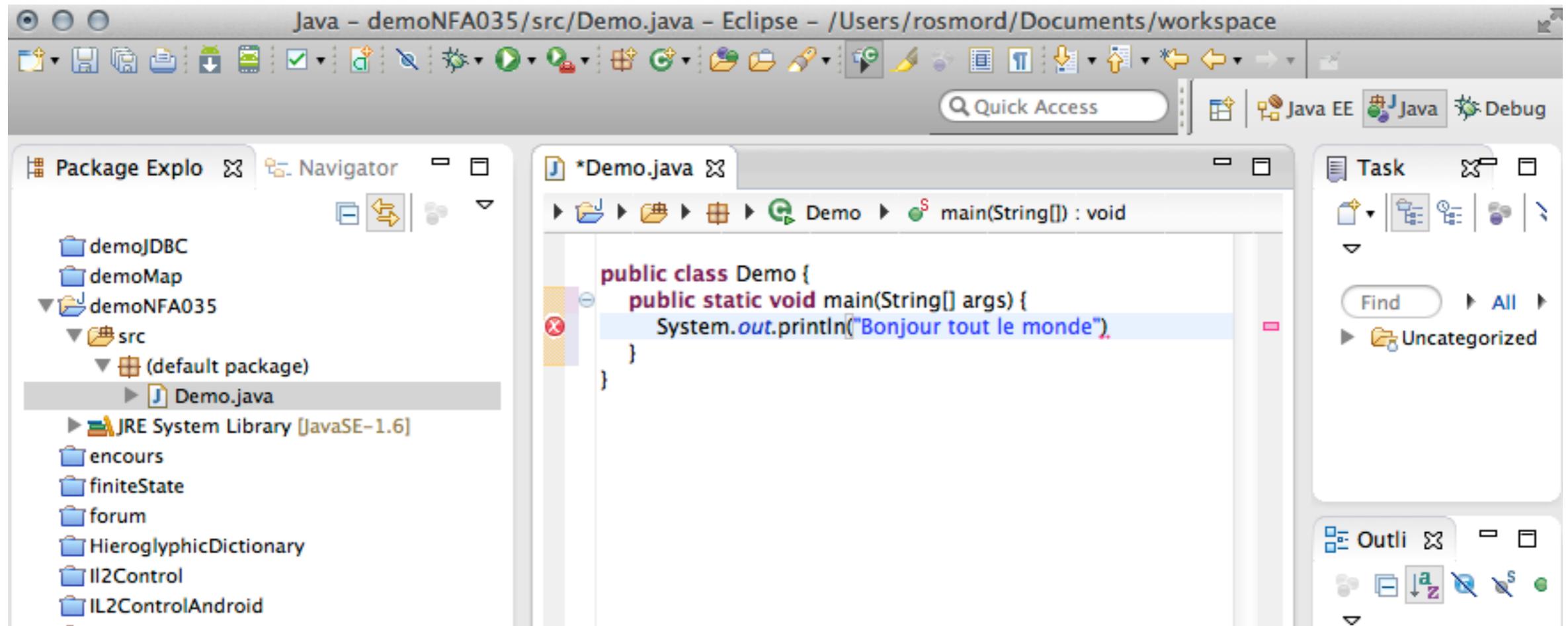
vue logique : « package explorer »



vue réelle (« Navigator »)

Créer une classe





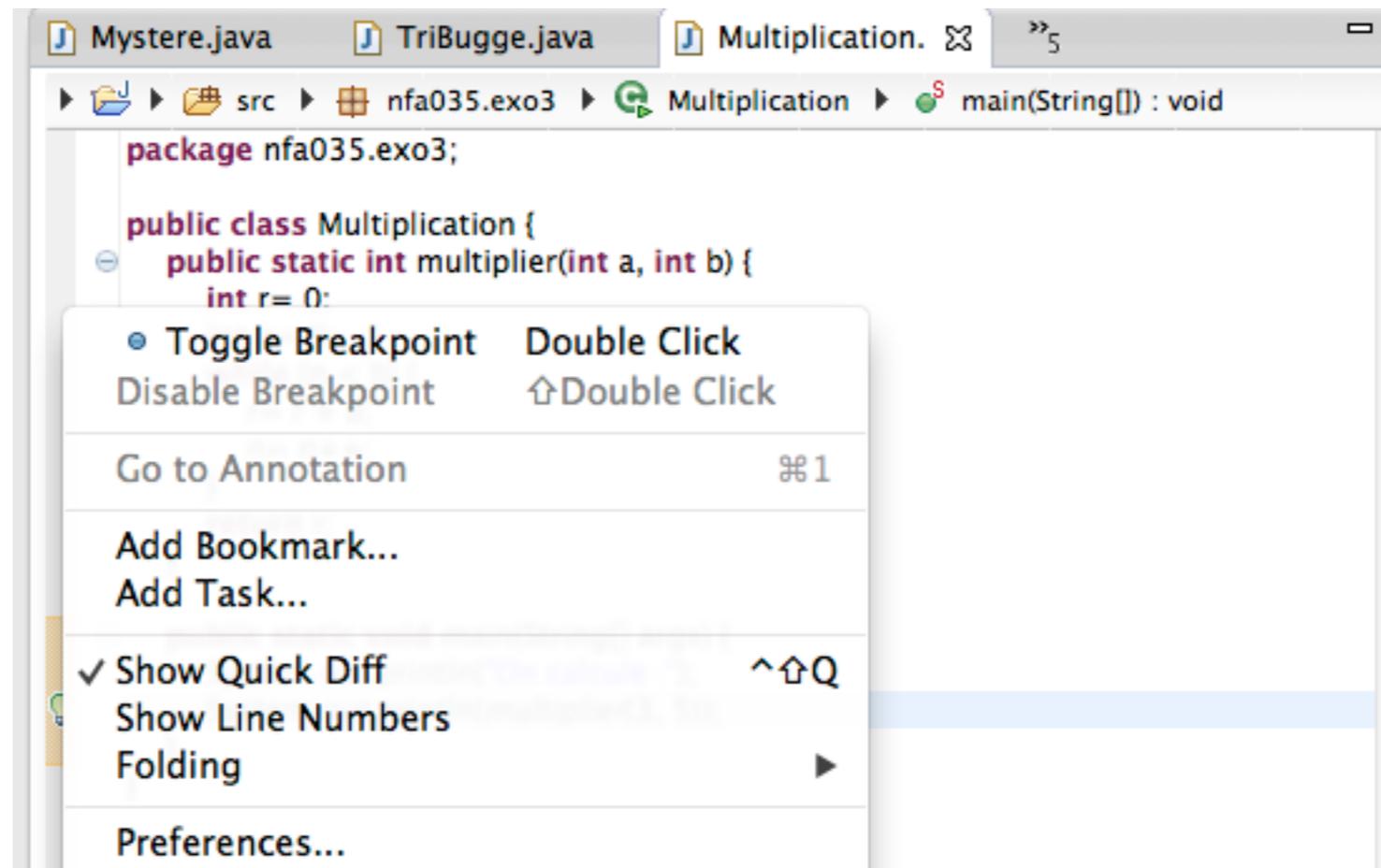
- le code est compilé « à la volée »
- les erreurs de syntaxe sont signalées en « temps réel ».

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...



Pile des appels

Valeurs des variables

The screenshot shows the Eclipse IDE interface during a debug session. The top toolbar contains various icons for running and debugging. The main workspace is divided into several panes:

- Debug Console:** Shows the call stack. The current frame is `Multiplication.multiplier(int, int) line: 5`. Other frames include `Multiplication.main(String[]) line: 16` and the JVM path `/System/Library/Java/JavaVirtualMachines/1.6.0.jc`.
- Variables View:** A table showing the current state of variables:

Name	Value
a	3
b	5
- Code Editor:** Displays the source code for `Multiplication.java`. The current line is `int r = 0;`. The code includes a `while` loop and a `return` statement.

```
package nfa035.exo3;

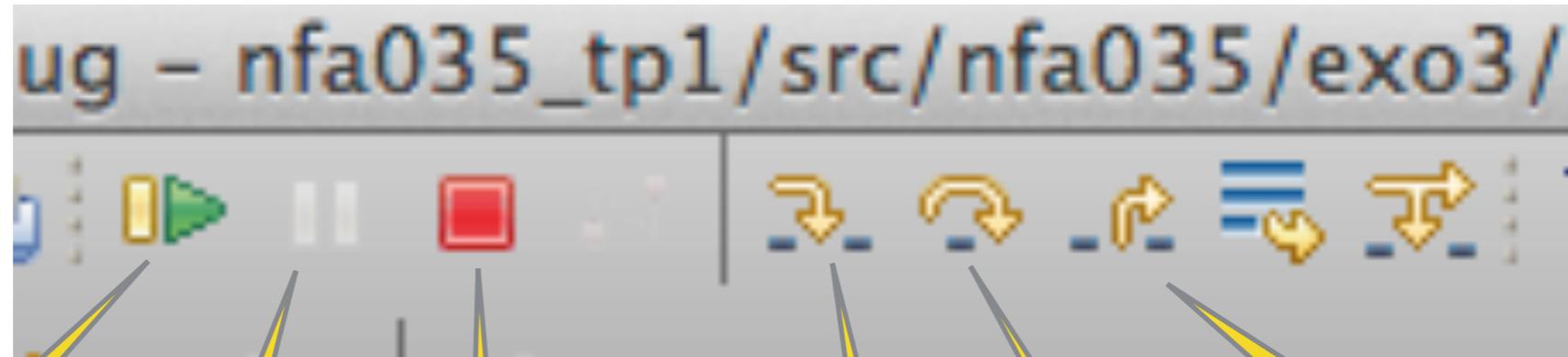
public class Multiplication {
    public static int multiplier(int a, int b) {
        int r = 0;
        int n = 0;
        while (n < b) {
            r = r + a;
            n = n++;
        }
        return r;
    }

    public static void main(String[] args) {
        System.out.println("On calcule :");
    }
}
```
- Outline View:** Shows the project structure with `nfa035.exo3` and `Multiplication` class, listing `main(String[]) : voi` and `multiplier(int, int)` methods.
- Console:** Shows the output of the program: `On calcule :`

ligne actuelle

code

Débugger : barre de commande



continuer l'exécution

pause

Tuer le programme

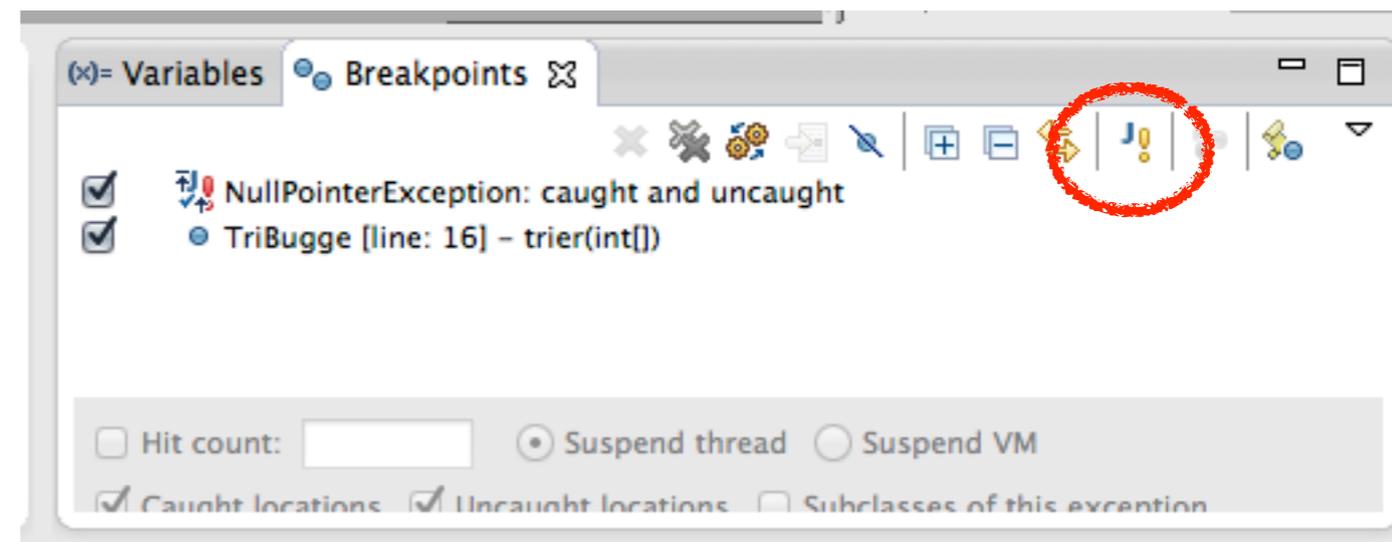
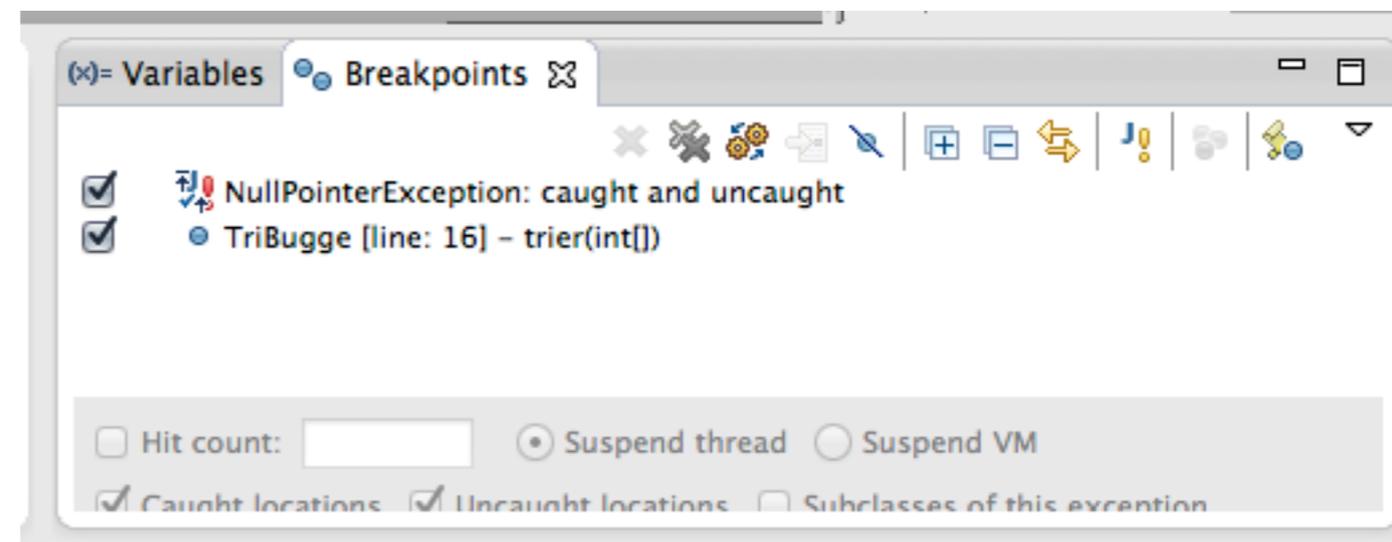
exécuter en « entrant » dans les fonctions

ligne à ligne

termine la méthode actuelle

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'appellent List...
- Comment les distinguer ?

The screenshot shows the Java Platform API Specification website. The left sidebar displays a list of classes under the heading "Packages". The class "Element" is highlighted in yellow, and there are five other instances of "Element" listed below it. The right sidebar shows the "Overview" tab selected, with a list of packages including "java.applet", "java.awt", "java.awt.color", "java.awt.datatransfer", "java.awt.dnd", "java.awt.event", "java.awt.font", and "java.awt.geom".

Java™ Platform
Standard Ed. 7

All Classes

Packages

java.applet

ECGenParameterSpec
ECKKey
ECPParameterSpec
ECPPoint
ECPPrivateKey
ECPPrivateKeySpec
ECPublicKey
ECPublicKeySpec
EditorKit
Element
Element
Element
Element
Element
ElementFilter
ElementIterator
ElementKind
ElementKindVisitor6
ElementKindVisitor7
Elements
ElementScanner6

Overview Package Class Use Tree

Prev Next Frames No Frames

Java™ Platform, Standard Edition 7 API Specification

This document is the API specification for the Java Platform, Standard Edition 7.

See: [Description](#)

Packages

Package

java.applet

java.awt

java.awt.color

java.awt.datatransfer

java.awt.dnd

java.awt.event

java.awt.font

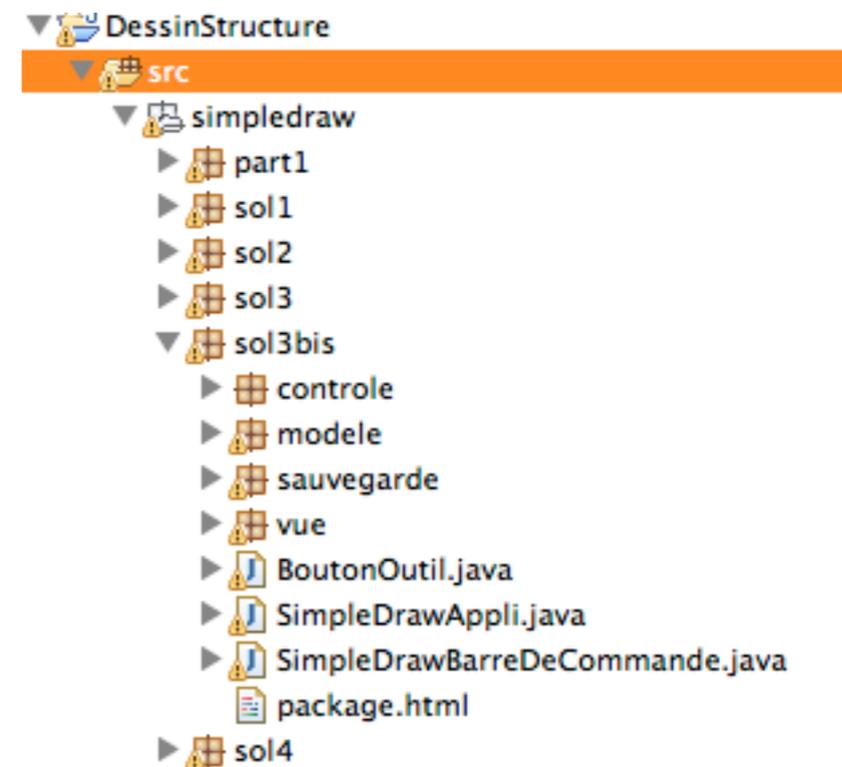
java.awt.geom

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

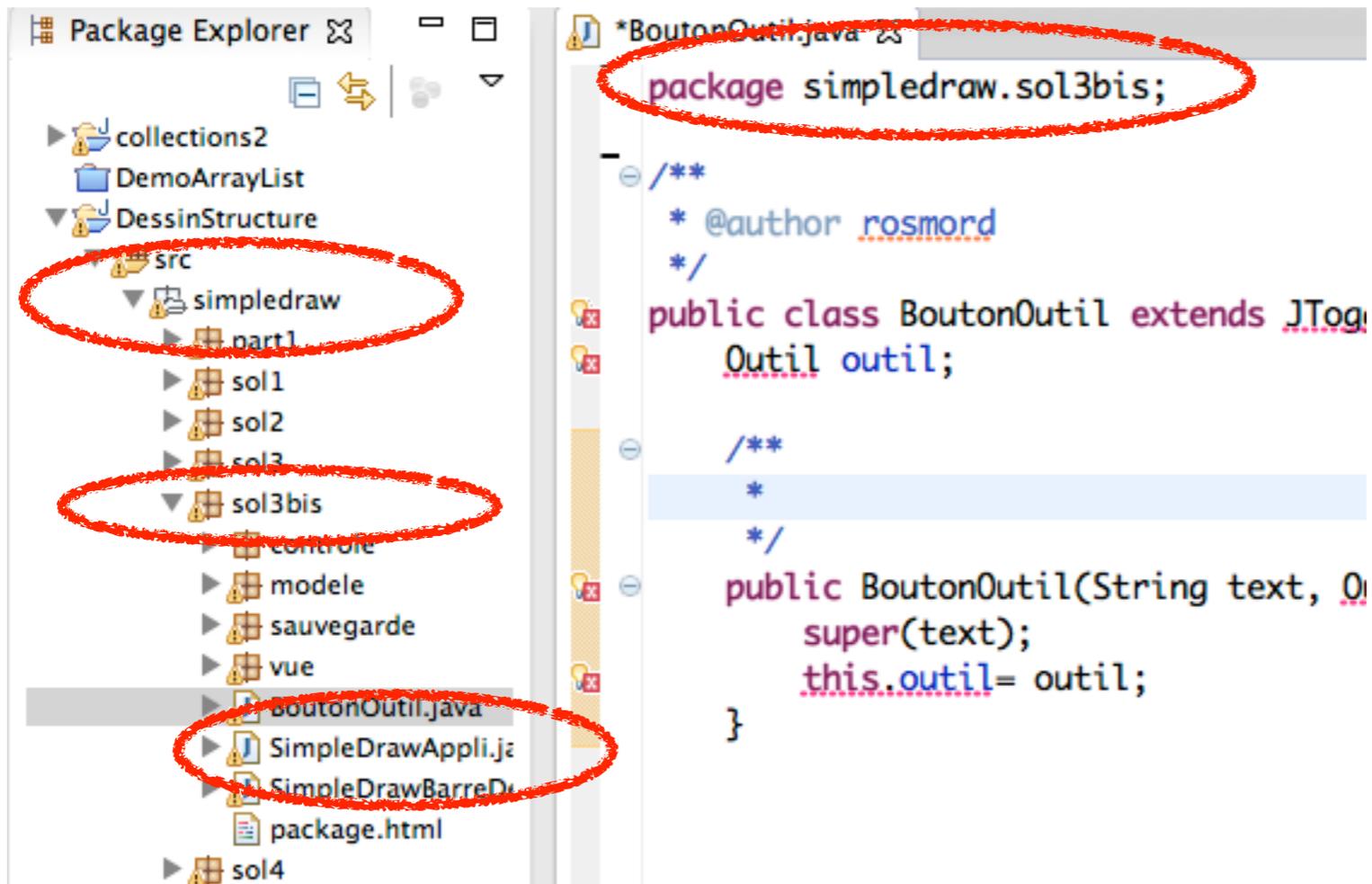
- Logiciel de dessin:
 - un package pour le « modele » (la représentation du dessin en mémoire)
 - un package pour la « vue » (son affichage)
 - un package pour le code de sauvegarde du dessin sur fichier



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 rosmond  
 */  
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

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

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 »