Eclipse et Java

Guide rapide d'utilisation d'un IDE Java

Ensimag 2A

Résumé

Un environnement de développement intégré (IDE - Integrated Development Environment) intègre un ensemble d'outils qui permettent d'accélérer le développement de logiciel. Il offre ainsi une interface graphique réunissant différents outils en un seul : un éditeur pour le code source, un compilateur, un débogueur pas à pas, des outils de tests de code, d'analyse du code source, ... L'objectif de ce guide TP est de découvrir ces fonctionnalités de base.

Ne pouvant couvrir tous les IDE existant pour java (intellij, visual studio code, netbeans, , ...), nous avons décidé de nous concentrer sur Eclipse . Vous retrouverez ces différentes fonctionnalités dans tout IDE.

1 Perspectives et vues

Lancez Eclipse. Une fois le workspace défini, Eclipse démarre dans la **perspective** par défaut "Java" (Menu `Window > Perspective > Open Perspective > Other > Java (Default)`). Qu'est ce qu'une perspective ?

Une perspective est un agencement de sous fenêtres composées de vues et des éditeurs. Eclipse offre plusieurs perspectives chacune spécialisée pour un traitement particulier (java, debogage, git, ...).

La figure suivante décrit la perspective par défaut "Java".

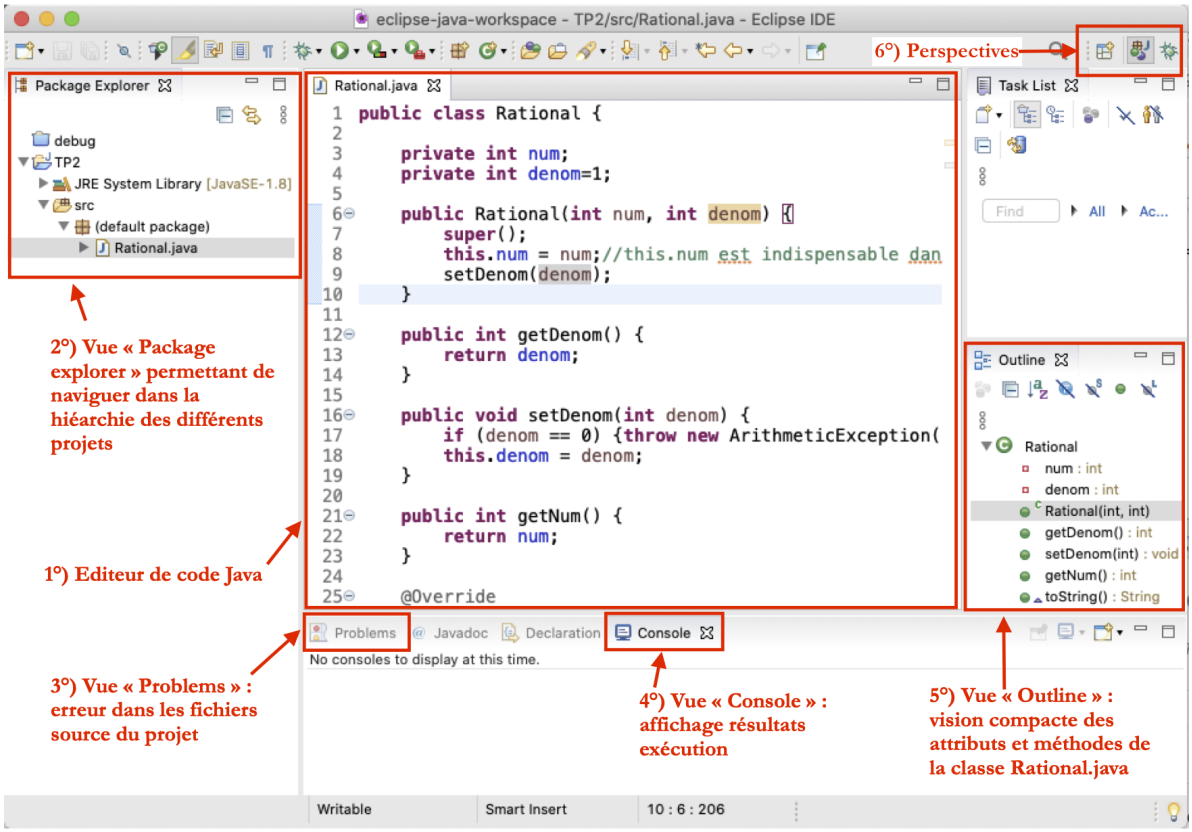


FIGURE 1 – Perspective "Java" par défaut d'Eclipse

Cette perspective est dédiée à l'édition de code et est contituée de :

- au centre : un éditeur de code java
- à gauche : une vue “package explorer” permettant de naviguer dans la hiérarchie des projets
- en bas :
  - la vue “Problems” qui résume les problèmes dans le code source
  - la vue “Console” propre à l’exécution qui permet de voir les données qui sont envoyées dans le flux standard de sortie et d’erreurs
- à droite : la vue “Outline” permettant de se déplacer rapidement dans le code du fichier `.java` en cours de traitement dans l’éditeur en offrant une vue compacte de ses attributs et méthodes.

Vous pouvez personnaliser/réorganiser une perspective en déplaçant/fermant les sous-fenêtres.

> **Info** : Afin de retrouver l’agencement par défaut de la perspective, utilisez le menu `Window > Perspective > Reset Perspective`

Reprenons maintenant le TP2 depuis le début mais en en utilisant cette fois-çi Eclipse.

2 Créer/Importer un projet

Créez le projet TP2 :

- Menu `File > New > Java Project`
- Project Name : TP2
- Bouton `Finish`

Créez la classe `Rational.java` :

- Clic droit sur projet → `File > New > Class`
- Name : Rational
- Bouton `Finish`

3 Génération de code

> **Info** : L’intérêt de la génération de code, en plus d’accélérer la production de code, est de se prémunir d’erreurs de syntaxe parfois longues à détecter pour un non-initié.

Ajoutez 2 attributs privés `num` et `denom` à la main à votre classe. Elle devrait ressembler à cela.

```
1  public class Rational {
2
3      private int num;
4      private int denom=1;
5
6  }
```

Vous avez alors la possibilité de générer du code automatiquement :

- 1 accesseur pour `num` et 1 accesseur/1 mutateur pour `denom` : Menu `Source > Generate Getters and Setters`, dérouler et cocher uniquement les options utiles
- constructeur `Rational(num, denom)` : Menu `Source > Generate Constructors using Fields`
- une methode `toString()` : Menu `Source > Generate toString()` dont le code par défaut sera surement à adapter

Il ne vous reste plus qu’à adapter le code pour coller avec les exigences de l’énoncé du TP (cf. suggestions commentaires dans le code ci-dessous).

```java
public class Rational {

    private int num;
    private int denom=1;

    public Rational(int num, int denom) {
        super();
        this.num = num;//this.num est indispensable dans ce cas de figure
        this.denom = denom;//à adapter en --> setDenom(denom);
    }

    public int getDenom() {
        return denom;
    }

    public void setDenom(int denom) {
        //adapter en traitant le cas où denom = 0
        //-->if (denom == 0) {throw new ArithmeticException("Division by zero...");}
        this.denom = denom;
    }

    public int getNum() {
        return num;
    }

    @Override
    public String toString() {
        return "Rational [getDenom()=" + getDenom() + ", getNum()=" + getNum() + "]";
        //à adapter en --> return this.getNum() + "/" + this.getDenom();
    }

}
```

4 Compilation et execution

4.1 Compilation

Eclipse ne propose pas de bouton pour compiler votre code car il le compile en permanence (Menu `Project > Build Automatically`) ce qui vous permet de voir les erreurs au fur et à mesure de la saisie de votre code. Eclipse indique de 3 manières différentes les erreurs de compilation :

- dans l'éditeur directement
  - un souligné rouge pour une erreur, jaune pour un warning sous le code posant problème
  - en marge de la ligne posant problème, une croix rouge pour les erreurs ( ) ou un panneau ( ) pour les warnings
  - dans les 2 cas, le survol par la souris de ces zones/icônes entraine l'ouverture d'une fenêtre précisant le problème et des solutions possibles
- dans la vue "Problems" (sous l'éditeur) où toutes les erreurs et warnings des projets ouverts apparaissent. Un double-clic sur un des problèmes vous amène à la ligne concernée.

Beaucoup de warning sont ignorés par défaut. Vous pouvez changer ce comportement via `Menu Window > Preferences > Java > Compiler > Errors/Warnings` sous Linux ou via `Preferences > Java > Compiler > Errors/Warnings` sous MacOs.

Un exemple de warning indiqué par défaut est une méthode portant le nom d'un contructeur. Vérifiez en ajoutant le code suivant à votre classe Rational. Pour rappel, un constructeur ne retourne rien donc ici le `void` est en trop.

```java
public void Rational() {} //une methode ne peut porter le nom d'un constructeur
```

4.2 Exécution

Créer la classe TestRational (à la création de la classe sélectionner les options pertinentes pour vous simplifier le travail, notamment pour avoir dès le départ une méthode `main`) :

```java
public class TestRational {
    public static void main(String[] args) {
        Rational r = new Rational(6, 4);
        System.out.println("r = " + r);
        }
}
```

Pour l'exécuter, clic-droit sur la classe `TestRational.java` dans la vue "Package explorer" et choisissez `Run as > Java Application`. Pour relancer l'exécution de TestRational, il suffit de cliquer sur l'icone  dans le bandeau supérieur.

4.3 A la ligne de commande

Attention : L'IDE vous masque beaucoup de choses (position réelles des fichiers source, bibliothèque, ...) en compilant et exécutant le code à votre place. Si votre projet ne compile/s'exécute plus dans votre IDE et que vous êtes dans l'incapacité de comprendre pourquoi, il est indispensable de pouvoir revenir aux outils en ligne de commande `javac` (compilateur) et `java` (interpréteur) afin de tirer cela au clair. Ce que vous avez dû faire pendant les deux premières séances...

Comment compiler et exécuter votre code en ligne de commande ? Il vous faut tout d'abord déterminer où se trouve le répertoire de votre projet. L'emplacement du workspace est créé par défaut dans `~/eclipse/workspace` sous MacOs et {~/eclipse-workspace} sous Linux.

Vous y trouverez un dossier TP2 avec l'arborescence suivante :

```
TP2
|-- bin <-- répertoire où sont stockés les .class
|   |-- Rational.class
|   |-- TestRational.class
|-- src <-- répertoire contenant les fichiers source du projet
    |-- Rational.java
    |-- TestRational.java
```

Pour compiler et éxécuter en ligne de commande TestRational et respecter l'arborescence par défaut des projets Eclipse, il faut :

- ouvrir votre terminal à la racine de votre projet : `<chemin_workspace>/TP2`
- pour compiler : `javac -d bin -sourcepath src -cp bin src/TestRational.java`
- pour exécuter : `java -cp bin TestRational`

Voire Fiche00-Introduction

Dans l'exemple ci-dessus, aucun paquetage n'est indiqué au début des classes `Rational.java` et `TestRational.java`. Dans la vue "Package Explorer", Eclipse fait apparaitre ces 2 classes sous un paquetage intitulé "default package" indiqué par l'icône 

.

Attention : Si vous ajoutez `package fr.ensimag.tp2;` au début des classes `Rational.java` et `TestRational.java`, elles appartiendront au paquetage `fr.ensimag.tp2`. Les `.class` seront positionnés alors dans le répertoire `TP2/bin/fr/ensimag/tp2` et les `.java` dans le répertoire `TP2/src/fr/ensimag/tp2`.
Pour compiler : `javac -d bin -sourcepath src -cp bin src/fr/ensimag/tp2/Rational.java`
Pour exécuter : `java -cp bin fr.ensimag.tp2.Rational`
Tous les détails dans la Fiche02-Paquetage du cours.

4.4 Archives JAR

Pour distribuer du bytecode Java, il peut être utile de fournir une simple archive plutôt qu'un ensemble de fichiers avec une arborescence compliquée. L'écosystème Java possède un outil pour ça, les fichiers JAR (pour Java ARchive).

Créons notre `TP2.jar` :

- Sélectionner dans la vue "Package Explorer" le projet
- Menu `File > Export > Java > JAR file`
- dans la boite de dialogue "JAR export"
  - décocher les fichiers `.classpath` et `.project` propre à l'environnement Eclipse

— `JAR file` : TP2.jar puis bouton `Next`
— bouton `Next` une 2ème fois
— `Main Class` : à l'aide du bouton `Browse`, sélectionnez le fichier `.class` qui sera exécuté par défaut

Il ne vous reste plus qu'à l'éxécuter : `java -jar TP2.jar`.

5 Complétion et Template

Vous l'avez remarqué, l'éditeur d'Eclipse offre la **coloration syntaxique**, l'**auto-indentation**, le **formatage**.

Il offre également la **complétion automatique**. Dans `TestRational.java`, si vous tapez "Ra" suivi des touches "Ctrl+Space", Eclipse vous suggèrera toutes les classes commençant par "Ra" dont Rational. De plus, si vous avez une référence "r" vers un objet Rational, tapez "r.get", Eclipse vous proposera toutes les méthodes de la classe Rational commençant par "get". Il ne vous reste plus qu'à choisir parmi les propositions.

Une autre bonne pratique est d'utiliser les **templates de code**. Ainsi, dans la méthode `main` de `TestRational.java`, si vous tapez "sysout" suivi de "Ctrl+Space", vous obtiendrez automatiquement `System.out.println`. Nombres de template sont déjà préconfigurés :

— "main" → `public static void main(String[] args){}`
— "for" → `for (int i = 0; i < array.length; i++){}`
— "if" → `if (condition){}`
— "pri" → `private`
— ...

Pour voir les templates existants ou créer votre propre template, rendez-vous au menu `Menu Window > Preferences > Java > Editor > Templates` sous Linux ou via `Preferences > Java > Editor > Templates` sous MacOs.

6 Déboguer

Mise en situation : vous arrivez en stage 2a et devez faire évoluer un code existant. Difficile de rentrer dans le code car il n'est pas documenté (*c'est mal !*). Pire : aucun test unitaire (nous verrons comment remédier à cela à la section 8). Vous faites quelques tests rapides pour voir si le programme fait ce qu'il est censé faire. Et là horreur ! Il est bogué... Heureusement, dans un recoin de votre mémoire, vous vous rappelez avoir utilisé en POO un outil particulièrement adapté à ce genre de situation : le débogueur.

Le **débogueur** permet :

— d'exécuter le code pas à pas (ligne par ligne) ou arrêter son exécution sur des breakpoints que vous aurez préalablement positionné aux endroits stratégigues ;
— d'avoir accès à la pile d'exécution ou aux variable en mémoire ;
— ...

Passons à la pratique. Selectionner tous les fichiers java de votre projet et effacez les. Remplacez-les par les fichiers `Rational.java` et `TestRational.java` contenus dans l'archive suivante : eclipse.tar.gz. Vous pouvez les glisser-déposer directement dans le projet Eclipse.

6.1 1er bogue : une boucle infinie !

A l'éxécution simple  , vous remarquez que le programme n'arrive pas au terme de son exécution et que rien ne s'affiche. Typique d'une boucle infinie. Afin de savoir où votre programme boucle :

— arrêtez le programme ()
— basculer dans la perspective "debug" via `Menu Window > Perspective > Open Perspective > Debug`
— Clic-droit sur `TestRational.java` et choisissez `Debug As > Java Application`
— suspendre l'exécution du programme ( ) grâce à la barre de contrôle disponible dans le bandeau supérieur.

A gauche dans la vue “debug”, vous pourrez double-cliquer dans la pile d’appels (vue “debug”) la ligne où votre programme boucle indéfiniment :

```
> TestRational (1) [Java Application]
    |-- TestRational at localhost:57398 (Suspended)
        |-- Thread [main] (Suspended)
            |-- Rational.gcd(int, int) line: 32 //<-- double-cliquez sur cette ligne
            |-- Rational.gcd(int, int) line: 36
            |-- Rational.gcd(int, int) line: 36
            |-- Rational.gcd(int, int) line: 36
            |-- TestRational.main(String[]) line: 17
        |-- Daemon System Thread [Signal Dispatcher] (Suspended)
        |-- Daemon System Thread [Finalizer] (Suspended)
        |-- Daemon System Thread [Reference Handler] (Suspended)
        |-- /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/...
```

Il ne vous reste plus qu’à :

— commenter la partie posant problème
— relancer l’exécution via le bouton ( )
— et vérifier que l’éxécution arrive bien à terme

Dans la vraie vie, la résolution d’un tel bogue nécessitera de relancer plusieurs fois l’exécution pas à pas en observant l’évolution des valeurs des variables à chaque boucle afin de trouver pourquoi votre code reste bloqué dans cette boucle. Ca tombe bien c est ce que nous allons faire dans la suite pour régler un 2ème bogue.

6.2 2ème bogue : un algorithme codé à la va-vite

Si vous regardez dans la vue “Console” le résultat de l’exécution, vous remarquez qu’un de vos tests sur le plus grand dénominateur commun à 6 et 4 ne fonctionne pas. Le résultat donné par la méthode `Rational.gcd2(4, 6)` à la ligne 12 de `TestRational.java` est faux.

Afin de trouver l’origine du problème :

— ajoutez un breakpoint à la ligne 12 de `TestRational.java` en double-cliquant à gauche du numéro de ligne (un point bleu apparait)
— exécutez `TestRational.java` en mode “debug” ( ).

Dans cette perspective “debug”, l’exécution de votre programme s’arrête ( ) à la ligne 12 de `TestRational.java`. Vous allez maintenant devoir exécuter le programme pas à pas jusqu’à tomber sur le bout de code posant problème. La barre de contrôle de l’exécution (détaillées dans la figure 2) offre différentes possibilités d’avancer dans l’exécution du code.

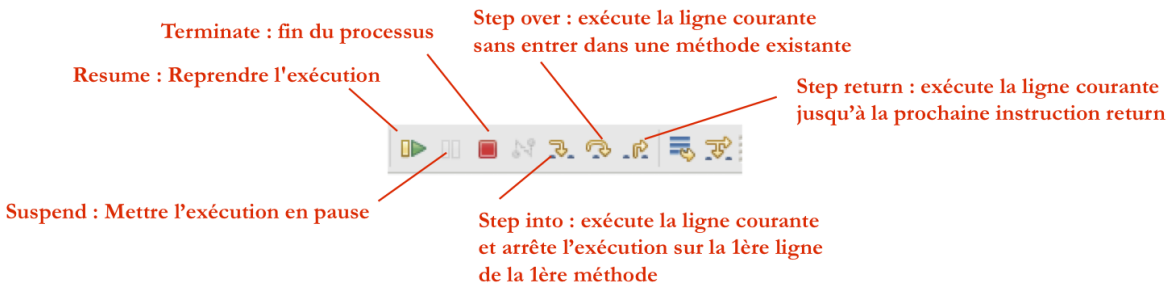


FIGURE 2 – Les boutons de contrôle de l’exécution

La source du problème se trouve en fait dans le code de la méthode `Rational.gcd2(int, int)`. Afin d’entrer dans le code de cette méthode, cliquez sur le bouton “Step into” .

Le programme s’arrête alors à la ligne 41 de `Rational.java` et votre perspective “debug” devrait ressembler à la figure suivante :

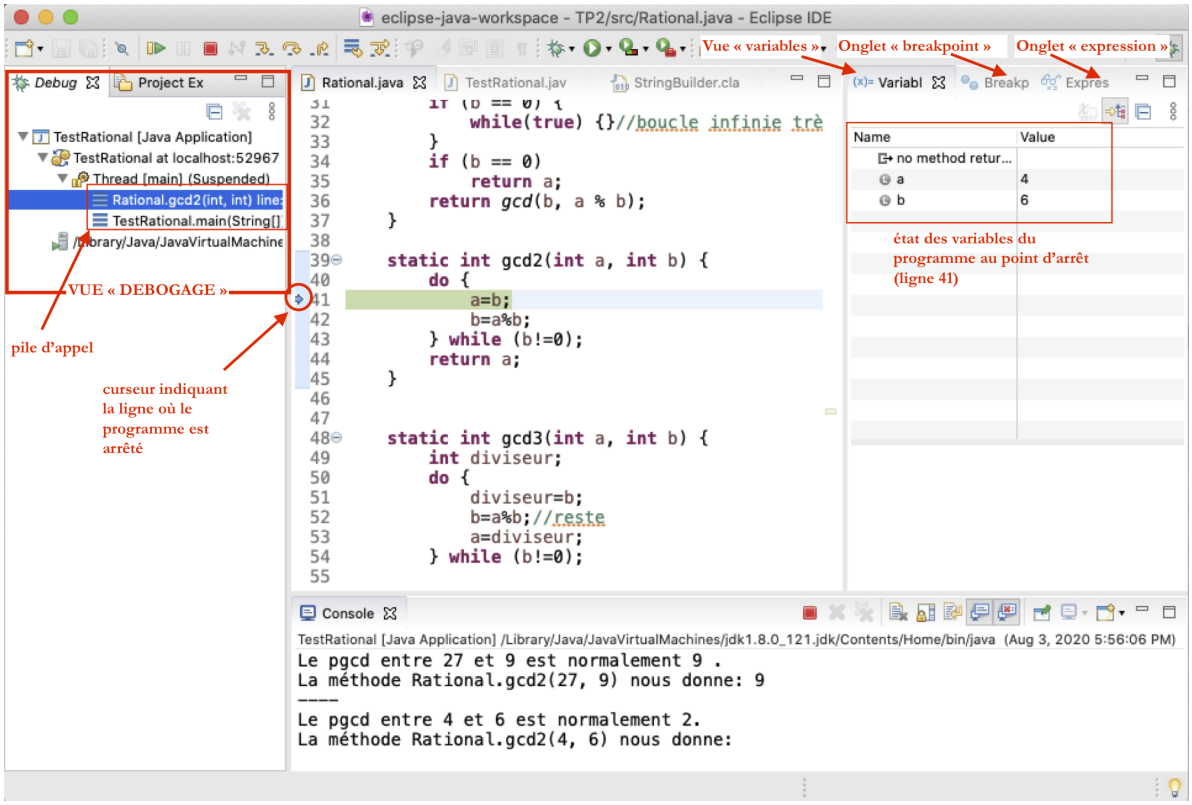


FIGURE 3 – perspective debug

Outre la vue "debug" à gauche, vous avez sur la droite la vue "Variables" qui vous permet de voir la valeur des variables à l'endroit où votre programme est actuellement arrêté.

- Pour l'instant, vous êtes à la ligne 41 et tout est normal ("a" vaut bien 4, "b" vaut 6)
- Continuer l'exécution de votre programme pas à pas ( )
- Vous voilà ligne 42 où la valeur de `a` est irrémédiablement écrasée par `b` menant à la valeur "6" fausse retournée à la ligne 44
- Résolvez le problème (si vous ne voyez vraiment pas, jetez un coup d'oeil à `Rational.gcd3()`) en répétant autant de fois que nécessaire cette exécution pas à pas
- Une fois le problème corrigé, vous pouvez revenir dans la perspective "Java" à l'aide du bouton  situé en haut à droite.

Documentation : Menu `Help > Help Contents` → Java development user guide > Getting Started > Basic tutorial > Debugging your programs

7 Refactoring

Votre chef de projet considère que le nom de la méthode `setDenom(int denom)` de votre classe `Rational.java` n'est pas suffisamment explicite. Il souhaite que vous la nommiez `setDenominateur(int denom)`.

Afin de gagner du temps, vous avez alors l'idée géniale d'utiliser la fonction pour rechercher le motif `setDenom` dans tout le code de votre projet et de le remplacer par `setDenominateur` (Menu `Search > Search` puis, en bas de la boite de dialogue "Search", cherchez "Replace…"). Cependant vous seriez vite amenés à repasser sur tous vos fichiers afin de corriger des remplacements inopinés et vérifier si les changements ont bien été propagés dans tous les fichiers du projet.

Eclipse facilite grandement ce genre d'opérations. Ainsi pour renommer une méthode (ou un attribut ou le nom d'une classe ou un paquetage …), il suffit d'un clic-droit sur la méthode `setDenom(int denom)` de la classe `Rational.java`, d'opter pour le menu `Refactor > Rename` et le renommer en `setDenominateur(int denom)` pour que cette modification soit propagée dans tous les fichiers utilisant cet attribut (Vérifiez par exemple dans `TestRational.main()`).

Plus généralement, l'ensemble des opérations permettant de retoucher votre code sans ajouter de fonctionnalités ou de bogues afin de rendre la code plus lisible et faciliter sa maintenance s'appelle le **refactoring**.

Eclipse intègre ainsi différentes actions pour réaliser ce refactoring. Le tableau ci-dessous offre une vue de quelques actions possibles :

Action	Description
Rename	Renomme l'élément selectionné dans tous les fichiers s'y référant
Move	Déplace l'élément sélectionné et corrige toutes les références à cet élément
Change method signature	Change les noms des paramètres d'une méthode, leur type, leur ordre et met à jour toutes les références à cette méthode
Extract Method	Crée une méthode à partir des instructions sélectionnées et remplace la sélection par un appel à cette méthode
Inline	Remplace par exemple une méthode par le corps de la méthode
Convert Anonymous Class to Nested	classes anonymes
...	...

Dans la suite, intéressons nous à l'action `extract method` particulièrement intéressante pour nettoyer les méthodes trop longues ou trop complexes.

Copiez les 2 méthodes suivantes à la fin de `Rational.java`. Votre esprit vif remarque rapidement que le code entre la ligne 8 et 13 de la méthode `mult(Rational r)` et entre la ligne 22 et 28 `add(Rational r)` est identique. Ce code permet de transformer la fraction obtenue en une fraction irréductible. Afin de simplifier la lecture de ce code, il serait intéressant de créer une méthode `simplify()` à partir de ce bout de code et d'y faire appel dans les 2 méthodes concernées.

```
1       /**
2        * Multiplies this with r, then simplifies the rational.
3        */
4       public void mult(Rational r) {
5           this.num = this.getNum() * r.getNum();
6           this.denom = this.getDenom() * r.getDenom();
7           int gcd = Rational.gcd(this.num, this.denom);
8           this.num /= gcd; //<--debut fonction simplify()
9           this.denom /= gcd;
10          if (this.denom < 0) {
11              this.num = -this.num;
12              this.denom = -this.denom;
13          }//<--fin fonction simplify()
14      }
15
16      /**
17       * Adds r to this, then simplifies the rational.
18       */
19      public void add(Rational r) {
20          this.num = r.getDenom() * this.getNum() + this.getDenom() * r.getNum();
21          this.denom = r.getDenom() * this.getDenom();
22          int gcd = Rational.gcd(this.num, this.denom);//<--debut fonction simplify()
23          this.num /= gcd;
24          this.denom /= gcd;
25          if (this.denom < 0) {
26              this.num = -this.num;
27              this.denom = -this.denom;
28          }//<--fin fonction simplify()
29      }
```

Pour ce faire :

— sélectionnez le code entre la ligne 8 et 13 de la méthode `mult(Rational r)`
— effectuez un clic-droit sur cette selection
— sélectionnez l'action `Refactor > Extract Method`
— dans la boite de dialogue "Extract Method",

— saisissez dans le champ “method name” `simplify`
— et cliquez le bouton `OK`

Que remarquez-vous ? Une nouvelle méthode `simplify()` a fait son apparition avec comme implémentation le corps du code sélectionné. La méthode `simplify()` a remplacé le code dans les méthodes `mult(Rational r)` et `add(Rational r)` rendant ainsi leur code plus lisible.

Par un clic-droit sur `simplify()` dans `add(Rational r)` et en choisissant l’action `inline`, vous effectueriez l’inverse de `extract method` en remplaçant `simplify()`par le corps de la méthode.

Documentation : Menu `Help > Help Contents` → Java development user guide > Reference > Refactoring > Refactor Actions

Pour aller plus loin : Si vous finissez en avance votre TPL, vous pourriez décider d’en profiter pour améliorer la lisibilité de votre code en mettant en pratique ces techniques de refactoring.

8 Test unitaire : Junit (S’il vous reste du temps !)

8.1 Ecrire des tests

Junit est un framework open source permettant de réaliser des tests unitaire du code Java. Il permet d’automatiser des tests et ainsi de s’assurer que le code répond à un ensemble de tests même après modifications.

Créons notre premier fichier de test `TestRationalJunit.java` :

— Menu `File > New > Junit Test Case`
  — Name : `TestRationalJunit.java`
  — Class under tests : Rational
  — Cliquez sur le bouton `Next`
  — Cochez les méthodes pour lesquelles vous souhaitez établir un test (`Rational(int,int)`, `setDenom()`, `toString()` et `gcd2(int, int)`)
  — Cliquez sur le bouton `Finish`
  — Cliquez `OK` pour l’ajout de la bibliothèque “Junit5” [1] au projet afin d’importer les classes nécessaires à la compilation de la classe `TestRationalJunit.java`

Afin d’éxécuter les tests automatiquement générés, il vous faut faire un clic-droit sur `TestRationalJunit.java` et `Run As > Junit Test`. Vous basculez dans la vue “Junit” composée de 2 fenêtres :

— l’une (en haut) résumant le résultat des tests effectués
— et l’autre (en bas) intitulée “Failure Trace” qui indique les erreurs éventuelles liées à un test sélectionné dans la fenêtre du haut

Tous vos tests doivent échouer puisque le code généré par défaut pour chaque test doit ressembler à :

```
1      @Test
2      void testTestSetDenom() {
3          fail("Not yet implemented");
4      }
```

Pour l’instant, La fenêtre “Failure Trace” indique pour chaque test le message suivant *org.opentest4j.AssertionFailedError : Not yet implemented*

Modifions maintenant l’implémentation des tests afin de vérifier si les méthodes vérifient bien ce qu’on attend d’elles :

1. Vous pourrez vérifier l’ajout de la librairie en question par un clic-droit sur le projet → `Properties > Java Build Path > Libraries`

```java
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class TestRationalJunit {

    Rational r = new Rational(6, 4);

    @Test
    void testRational() {
        Rational rational = new Rational(8, 3);
        assertEquals(8, rational.getNum());
        assertEquals(3, rational.getDenom());
    }

    @Test
    void testSetDenom() {
        r.setDenom(1);
        assertEquals(1, r.getDenom());
    }

    @Test
    void testToString() {
        assertEquals("6/4", r.toString());
    }

    @Test
    void testGcd2() {
        assertEquals(9, Rational.gcd2(27, 9),"pgcd de 27 et 9");
        assertEquals(2, Rational.gcd2(4, 6),"pgcd de 4 et 6");
    }

}
```

Relancez les tests via le bouton  .

Seul le test `testGcd2()` échoue. Double-cliquez sur ce test. L'éditeur vous amènera à la ligne 31 de `TestRational.java` qui pose problème. Pour rappel, la méthode gcd2(int, int) est erronée (cf. section précédente "Debug"). Remplacez `Rational.gcd2(4, 6)` par `Rational.gcd3(4, 6)` et relancez uniquement ce test (Clic-droit sur ce test → `Run`).

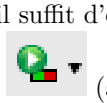
Vous trouverez la liste des assertions disponibles dans la javadoc associée.

Enfin il est également possible de tester les exceptions. Ajoutez le test suivant :

```java
    @Test
    void testException() {
        Exception exception = assertThrows(ArithmeticException, () ->
        r.setDenom(0));
        assertEquals("Division by zero...", exception.getMessage());
    }
```

Remplacez `ArithmeticException` par `IOException.class` et vous verrez que le test ne passe plus. En effet, `ArithmeticException` hérite de `RuntimeException` et non de `IOException` (cf. Fiche04-Exception → Section "Hiérarchie de classes des exceptions"). Idem si vous modifiez le message d'exception "Division by zero...".

8.2 Couverture de code

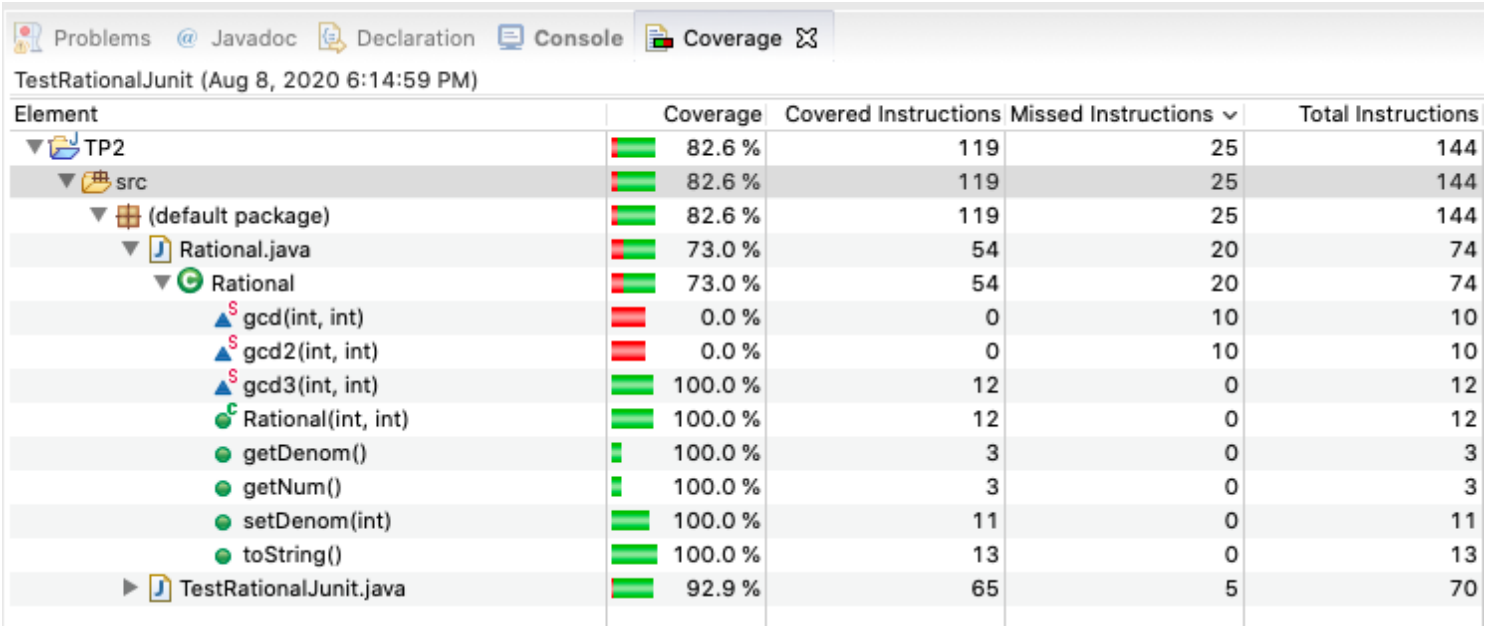
Afin de tester la couverture de votre code, il suffit d'effectuer un clic-droit sur `TestRationalJunit.java` et de choisir `Coverage As > Junit Test` ou de cliquez sur le bouton  (avec `TestRational.java` dans l'éditeur).

Votre code dans `Rational.java` se colore en différentes couleurs :

— vert : les parties du code testées

— rouge : les parties non couvertes
— jaune : les parties partiellement couvertes (une condition par exemple)

Un onglet “Coverage” fait son apparition en bas affichant un rapport de couverture de code.



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
TP2	82.6 %	119	25	144
src	82.6 %	119	25	144
(default package)	82.6 %	119	25	144
Rational.java	73.0 %	54	20	74
Rational	73.0 %	54	20	74
gcd(int, int)	0.0 %	0	10	10
gcd2(int, int)	0.0 %	0	10	10
gcd3(int, int)	100.0 %	12	0	12
Rational(int, int)	100.0 %	12	0	12
getDenom()	100.0 %	3	0	3
getNum()	100.0 %	3	0	3
setDenom(int)	100.0 %	11	0	11
toString()	100.0 %	13	0	13
TestRationalJunit.java	92.9 %	65	5	70

FIGURE 4 – Rapport de couverture de code

Le but du jeu ensuite serait de faire en sorte que vos tests couvrent 100% du code.

Documentation : Menu `Help > Help Contents` → EclEmma Java Code Coverage > User Guide