Aix*Marseille université
Socialement engagée

Année 2021-22

Domaine Sciences et Technologies
Licence informatique

Programmation 2 : TP 2 (informatique)
Code UE : SIN3U02L

Affichage de l'ensemble de Mandelbrot

1 Affichage de l'ensemble de Mandelbrot

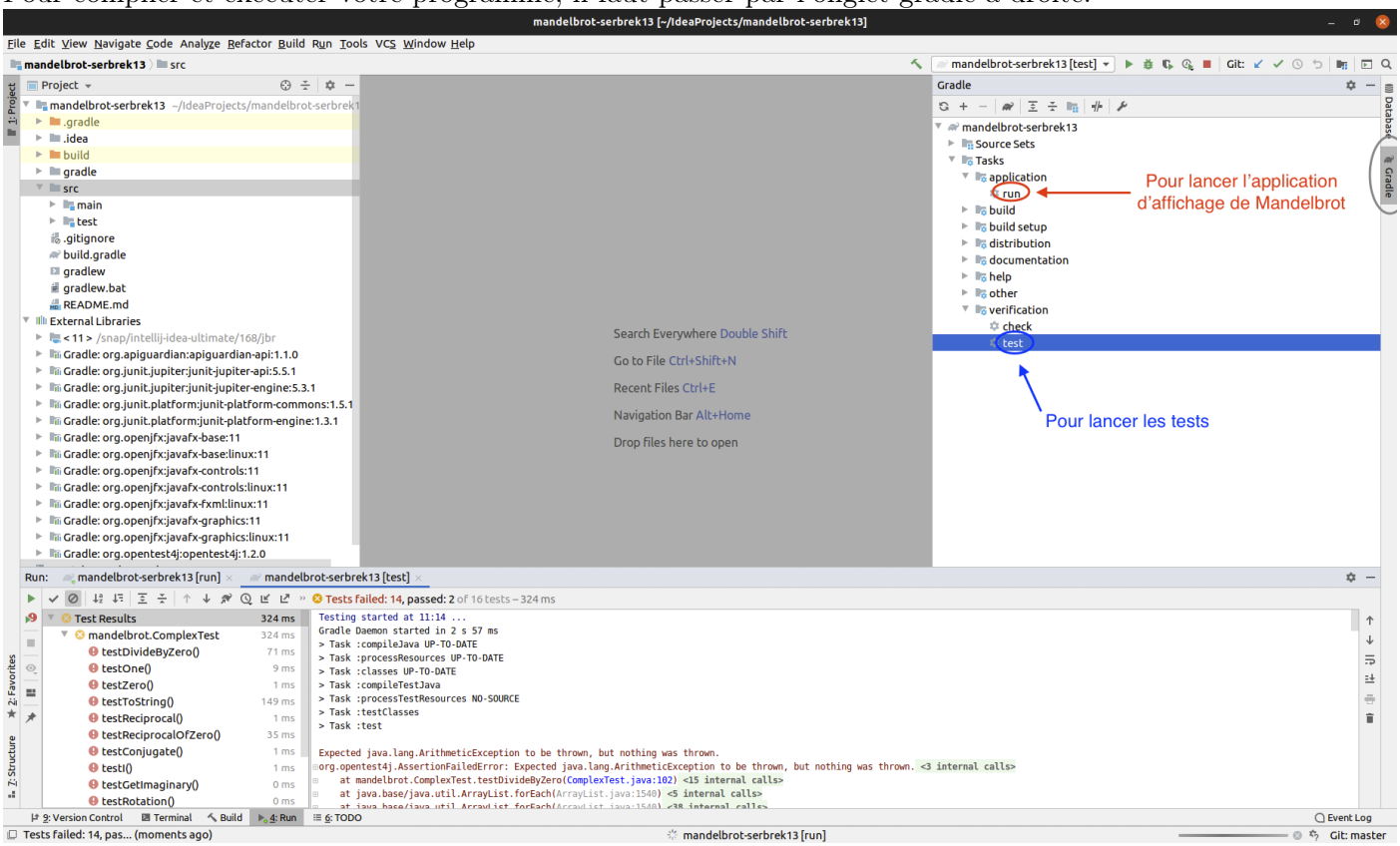
On va travailler sur ce TP sur l'affichage de l'ensemble de Mandelbrot. Pour cela, on va utiliser un projet pré-existant. Malheureusement la classe `Complex` de ce projet est bourrée d'erreurs. Le but du TP sera donc de corriger la classe `Complex` en s'aidant de tests unitaires.

1.1 Récupérer le dépôt

Comme pour le premier TP, on va utiliser git pour la gestion de versions. Il vous faut donc vous reporter aux consignes du précédent TP. Le lien vers le projet à forker est le suivant : lien.

1.2 Exécuter le projet du dépôt

Pour compiler et exécuter votre programme, il faut passer par l'onglet gradle à droite.



— pour les tests il faut cliquer deux fois sur `mandelbrot` -> `Tasks` -> `verification` -> `test`. Pour le moment, les tests ne passeront pas car certaines classes sont incomplètes.
— pour exécuter l'application qui est censé afficher la fractale de Mandelbrot, il faut cliquer deux fois sur `mandelbrot` -> `application` -> `run`. Vous devriez obtenir l'affichage suivant au bout de quelques minutes (le calcul prend du temps).

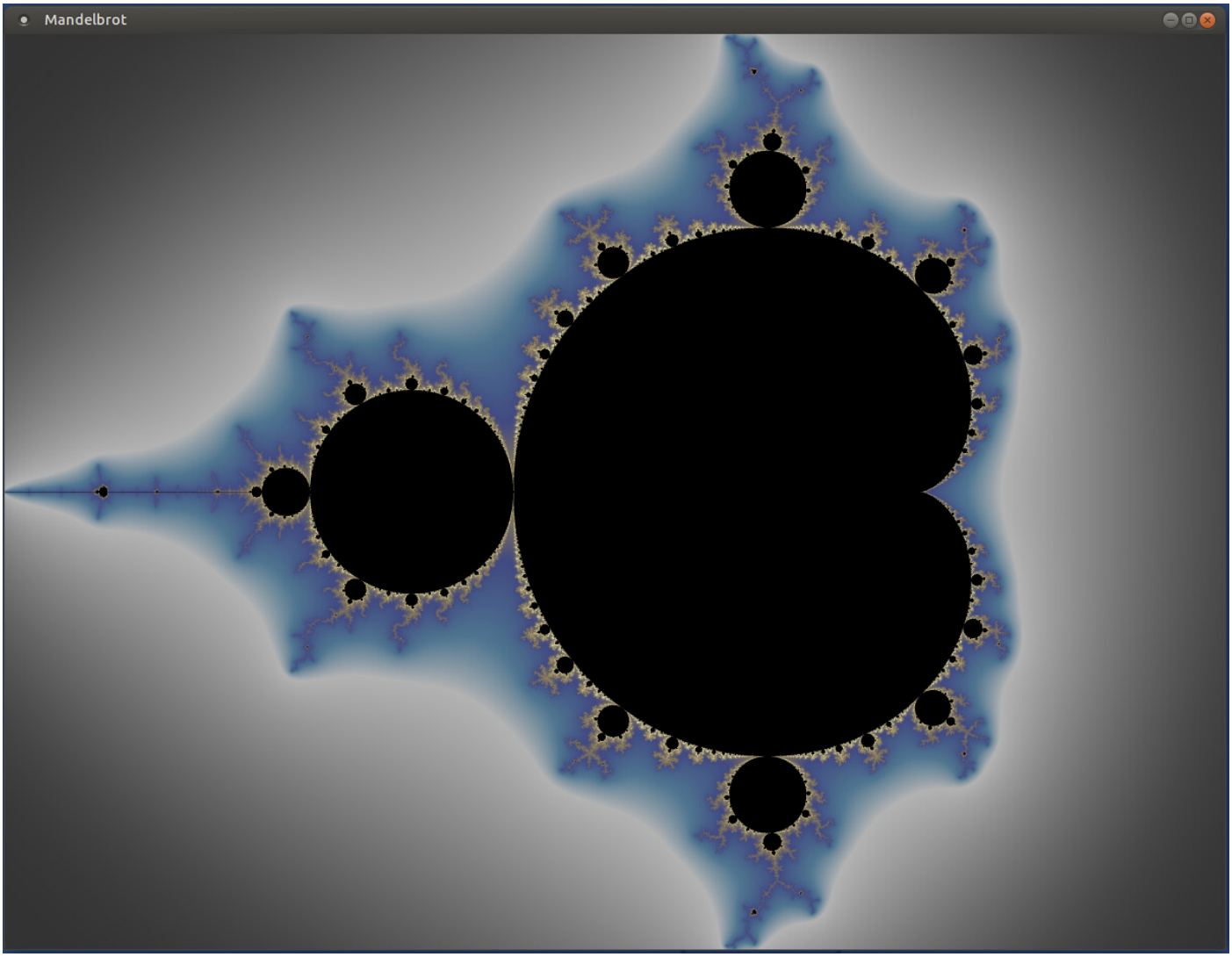
1.3 Consignes pour le début du TP

Modifiez le fichier `README.md`. Mettez votre nom, votre **numéro de groupe** ainsi que le nom et le **numéro de groupe** de votre éventuel co-équipier. Faites un `commit` avec pour message "inscription d'un membre de l'équipe", puis un `push`.

2 Tâches à effectuer

On cherche à corriger la classe `Complex.java` afin d'afficher correctement l'ensemble de Mandelbrot. Vous ne devez modifier que la classe `Complex.java` (sauf pour les tâches optionnelles). La version de base du projet produit l'affichage suivant :

L'objectif du TP est d'obtenir l'affichage correct suivant :



Toutes les méthodes/classes/constructeurs/initialisation des constantes sont erronées à l'exception de la méthode `int hashCode()`.

2.1 Pourquoi des tests ?

Les calculs nécessaires à l'affichage de l'ensemble de Mandelbrot prennent plusieurs minutes alors que des tests unitaires peuvent se lancer en quelques secondes. Il est donc bien plus efficace de tester les méthodes et de ne lancer le calcul pour l'affichage que lorsque tous les tests sont passés avec succès. De plus, le résultats des tests avec les valeurs attendues donne une bien meilleure indication des erreurs que l'affichage de l'ensemble de Mandelbrot qui est dur à analyser.

2.2 Tâche 1 : corriger les méthodes pour lesquelles les tests existent

Pour cette tâche, vous devez corriger les constructeurs, méthodes et des initialisations des constantes afin qu'ils passent les tests déjà écrits dans la classe `ComplexTest` se trouvant dans le répertoire `src/test/java/mandelbrot/` :

— `Complex(double real, double imaginary)` (constructeur)
— `double getImaginary()` (méthode)
— `double getReal()` (méthode)
— `static Complex ZERO` (constante)

— `static Complex ONE` (constante)
— `static Complex I` (constante)
— `Complex negate()` (méthode)
— `Complex reciprocal()` (méthode)
— `Complex divide(Complex divisor)` (méthode)
— `Complex conjugate()` (méthode)
— `static Complex rotation(double radians)` (méthode)

Pour trouver et corriger les erreurs dans le programme, vous devrez lancer les tests. Vous devez faire un commit à chaque fois que vous corrigez un élément du programme : constructeur, méthode ou initialisation d'une constante. Si vous souhaitez utiliser dans vos test une fonction qui n'est pas testée, il est plus que conseillé de la tester au préalable. Il est conseillé de ne faire des push que lorsque les tests passent. En effet, le projet est configuré pour lancer des tests sur le serveur et vous recevrai donc un mail de celui-ci si les tests ne passent pas.

En ce qui concerne les opérations sur les complexes, vous vous fierez à la section sur les opérations élémentaires de la page wikipedia sur les nombres complexes.

Vous pouvez vous référez à la documentation de la classe Complex pour la corriger ainsi qu'aux explications sur les tests unitaires à la fine de cette planche de TP.

2.2.1 Attention

Pour corriger ces méthodes, vous avez peut-être besoin de corriger d'autres méthodes.

2.3 Tâche 2 : écrire les tests puis corriger les méthodes pour lesquelles les tests n'existent pas

Pour cette tâche, vous devez corriger les méthodes suivantes (si vous ne l'avez pas déjà fait) :
— `public static Complex real(double real)`
— `public Complex add(Complex addend)`
— `Complex subtract(Complex subtrahend)`
— `Complex multiply(Complex factor)`
— `double squaredModulus()`
— `double modulus()`
— `Complex pow(int p)`
— `Complex scale(double lambda)`

Pour trouver et corriger les erreurs dans le programme vous devrez écrire des méthodes de test dans la classe `ComplexTest` en vous inspirant des tests déjà écrits pour les autres méthodes. Vous devez faire un commit à chaque fois que vous corrigez un méthode.

3 Tests unitaires

— Tester seulement une unité du programme : classe, méthodes, ...
— Vérifier un comportement :
  — cas normaux
  — cas limites
  — cas anormaux

3.1 Assertions utiles en JUnit/assertJ

Pour les tests unitaires de la classe `Complex`, on va utiliser JUnit 5 et AssertJ.

`JUnit` est le framework le plus répandu pour les tests unitaires en java alors qu'`AssertJ` est une bibliothèque permettant de rendre les tests plus lisible en définissant des assertions.

Pour que les méthodes et annotations de tests soient accessibles, il faut mettre les `import` suivants au début du fichier :

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;
```

Une méthode de test doit avoir la forme suivante :

```
@Test
  void testNameOfTheMethodTested(){
    // Assertions
  }
```

Vous pouvez vous inspirez des tests déjà écrits ainsi que des explications ci-dessous pour écrire vos propres tests :

- `assertThat(condition).isTrue()` : vérifie que le booléen `condition` est vrai.
- `assertThat(condition).isFalse()` : vérifie que le booléen `condition` est faux.
- `assertThat(actual).isEqualTo(expected)` : vérifie que `expected` est égal à `actual` (égal : `equals` pour les objets et `==` pour les types primitifs).
- `assertThat(actual).isCloseTo(expected, within(delta))` : vérifie que les trois doubles `expected`, `actual` et `delta` respectent l'inégalité $|expected - actual| \leq delta$. Il est important de tester les doubles avec une marge d'erreur possible car par exemple le test `assertThat(0.1+0.1+0.1).isEqualTo(0.3)` ne passe pas.
- `assertThat(object).isNull()` : vérifie que la référence `object` est `null`
- `assertThat(object).isNotNull()` : vérifie que la référence `object` **n'**est **pas** `null`
- `assertThat(actual).isSameAs(expected)` : vérifie que les deux objets `expected` et `actual` sont les mêmes instances (la même référence).
- `assertThat(list).containsExactly(e1, e2, e3)` : vérifie que la liste `list` contient uniquement les éléments `e1`, `e2` et `e3` dans cet ordre.
- `assertThat(list1).containsExactlyElementsOf(list2)` : vérifie que les deux listes `list1` et `list2` contiennent les mêmes éléments dans le même ordre.
- `fail(String message)` : échoue toujours en affichant le message en paramètre (utile pour un test pas encore fini).

Il est possible de provoquer l'affichage d'un `message` lors d'un test faux en appelant `as(message)` sur le retour d'un `assertThat` comme l'exemple ci-dessous.

```
TolkienCharacter frodo = new TolkienCharacter("Frodo", 33, Race.HOBBIT);
assertThat(frodo.getName()).as("check name")
                           .isEqualTo("Frodo");
assertThat(frodo.getAge()).as("check age")
                           .isEqualTo(33);
```

On peut enchaîner les assertions comme ci-dessous :

```
assertThat(frodo.getName())
                .isEqualTo("Frodo")
                .isNotEqualTo("Frodon");
```

Vous trouverez davantage d'exemples d'utilisation des tests au lien suivant : AssertJ Core.

4 Tâches optionnelles

Quelques tâches possibles mais optionnelles :

— Ajouter dans l'interface graphique une manière pour l'utilisateur de définir la région à afficher.

— Ajouter dans l'interface graphique une manière pour l'utilisateur de personnaliser les couleurs d'affichage de l'ensemble.