TD1 : Tests Unitaires avec JUnit 5

Objectifs pédagogiques :

- Apprendre à écrire et exécuter des tests unitaires avec **JUnit 5**.
- Comprendre les concepts de base des tests unitaires, y compris les annotations et les assertions.
- Améliorer la qualité du code en introduisant des tests automatisés.

Exercice 1 : Premier test unitaire avec JUnit 5

But : Créer et exécuter un test unitaire simple avec JUnit 5.

Instructions :

1. Créez une classe Java Calculatrice avec une méthode addition(int a, int b) qui retourne la somme de a et b.
2. Créez un test unitaire dans une classe CalculatriceTest pour tester la méthode addition().
3. Utilisez l'annotation @Test pour marquer la méthode de test et assertEquals() pour vérifier que l'addition de 2 et 3 renvoie 5.
4. Exécutez les tests avec JUnit 5 et vérifiez que le test passe.

Exercice 2 : Test d'égalité avec assertEquals()

But : Utiliser l'assertion assertEquals() pour vérifier l'égalité des résultats.

Instructions :

1. Dans votre classe CalculatriceTest, ajoutez plusieurs tests pour vérifier la méthode addition() avec différents ensembles de données (par exemple, 1+2, -1+2, etc.).
2. Utilisez assertEquals(expected, actual) pour comparer le résultat attendu avec le résultat réel.
3. Vérifiez que tous les tests passent.

Exercice 3 : Tests avec des valeurs limites

But : Tester les limites des valeurs dans votre fonction.

Instructions :

1. Modifiez la méthode addition() pour tester les cas limites, comme l'addition de nombres très grands (par exemple, Integer.MAX_VALUE).
2. Ajoutez un test unitaire pour vérifier que l'addition de deux grands nombres ne dépasse pas la limite.
3. Utilisez assertTrue() pour vérifier que le résultat est inférieur à une certaine limite.

Exercice 4 : Tests d'exception avec assertThrows()

But : Tester les exceptions générées par des méthodes.

Instructions :

1. Modifiez la méthode addition() pour qu'elle lance une exception (ArithmeticException) lorsque le résultat dépasse une certaine valeur (par exemple, Integer.MAX_VALUE).
2. Créez un test unitaire pour vérifier que l'exception est correctement levée lorsqu'une telle situation se produit.
3. Utilisez l'assertion assertThrows() pour tester l'exception.

Exercice 5 : Utilisation de @BeforeEach pour initialisation

But : Apprendre à initialiser des objets avant chaque test avec @BeforeEach.

Instructions :

1. Créez une classe Calculatrice avec plusieurs méthodes (addition(), soustraction(), etc.).
2. Utilisez l'annotation @BeforeEach pour initialiser un objet Calculatrice avant chaque test.
3. Écrivez plusieurs tests pour les méthodes addition() et soustraction(), en vous assurant que chaque test utilise un objet fraîchement initialisé.

Exercice 6 : Utilisation de @AfterEach pour nettoyage

But : Effectuer un nettoyage après chaque test avec @AfterEach.

Instructions :

1. Dans la même classe de test que dans l'exercice précédent, ajoutez une méthode annotée avec @AfterEach.
2. Cette méthode doit afficher un message indiquant que le nettoyage est effectué après chaque test.
3. Vérifiez que le message s'affiche après chaque exécution de test.

Exercice 7 : Tests paramétrés avec @ParameterizedTest

But : Tester des méthodes avec différents jeux de données en utilisant @ParameterizedTest.

Instructions :

1. Créez un test paramétré pour tester la méthode addition() avec différentes valeurs d'entrée.
2. Utilisez l'annotation @ValueSource pour fournir une liste de valeurs.
3. Faites en sorte que le test vérifie que l'addition de différentes paires de nombres (par exemple, 1+1, 2+2, etc.) fonctionne correctement.

Exercice 8 : Utilisation de @DisplayName pour personnaliser les tests

But : Ajouter un nom personnalisé aux tests avec l'annotation @DisplayName.

Instructions :

1. Dans votre classe de test, ajoutez l'annotation @DisplayName à vos méthodes de test.
2. Utilisez des noms explicites pour chaque test afin d'améliorer la lisibilité des rapports de test (par exemple, @DisplayName("Test de l'addition de nombres positifs")).
3. Exécutez les tests et vérifiez que les noms des tests sont affichés correctement dans les résultats.

Exercice 9 : Tests de la méthode toString()

But : Tester une méthode toString() pour vérifier sa bonne implémentation.

Instructions :

1. Créez une classe Personne avec des attributs nom et age.
2. Implémentez une méthode toString() qui retourne une chaîne de caractères décrivant la personne (par exemple, "Nom: Jean, Age: 30").
3. Écrivez un test unitaire pour vérifier que la méthode toString() renvoie la chaîne correcte.
4. Utilisez assertEquals() pour vérifier la valeur retournée.

Exercice 10 : Test de la classe List avec assertAll()

But : Tester plusieurs assertions dans un même test avec assertAll().

Instructions :

1. Créez une classe ListeUtil avec une méthode ajouterElement(List<String> list, String element) qui ajoute un élément dans une liste.
2. Créez un test unitaire pour vérifier que l'élément est ajouté à la liste.
3. Utilisez l'assertion assertAll() pour tester plusieurs conditions dans le même test, comme vérifier la taille de la liste et la présence de l'élément ajouté.

Exercice 11 : Vérification d'une méthode de validation d'email

Crée une classe Utilisateur avec une méthode estEmailValide(String email), qui retourne true si l'email respecte un format valide (ex: "exemple@test.com"), sinon false.

Écris des tests unitaires avec JUnit pour cette méthode, en testant des emails valides et non valides.

Exercice 12 : Vérification d'une liste de noms

Crée une méthode obtenirNomsUtilisateurs() qui retourne une liste de noms d'utilisateurs.

Écris un test JUnit pour vérifier que :

1. La liste retournée n'est pas null.
2. La liste contient bien un certain utilisateur (ex: "Alice").

Exercice 13 : Test d'une méthode de concaténation de chaînes

Crée une méthode concatener(String a, String b) qui retourne la concaténation des deux chaînes.

Écris un test unitaire pour vérifier que la concaténation fonctionne bien, même si l'un des paramètres est une chaîne vide.

Exercice 14 : Vérification d'un utilisateur actif

Crée une classe Utilisateur avec un champ actif (boolean) et une méthode estActif().

Ajoute un test JUnit pour vérifier qu'un utilisateur nouvellement créé est bien actif.

Exercice 15 : Test d'une méthode de recherche dans une liste

Crée une classe GestionProduits avec une méthode rechercherProduit(String nomProduit) qui retourne true si le produit existe dans la liste, sinon false.

Ajoute des tests JUnit pour vérifier les cas où :

1. Le produit est présent.
2. Le produit est absent.
3. La recherche est faite avec une chaîne vide.

Exercice 16 : Vérification d'une exception sur une méthode de suppression

Ajoute une méthode supprimerUtilisateur(String id) qui lève une exception IllegalArgumentException si l'ID est null ou vide.

Écris un test JUnit pour vérifier que l'exception est bien levée dans ces cas.

Exercice 17 : Test d'une méthode de transformation de texte

Crée une méthode transformerTexte(String texte) qui met en majuscules toutes les lettres et supprime les espaces en début et fin.

Écris des tests JUnit pour vérifier son bon fonctionnement.

Exercice 18 : Test d'un service de génération d'identifiant

Crée une classe GenerateurID avec une méthode generer() qui retourne un identifiant aléatoire unique de 10 caractères.

Ajoute un test unitaire pour vérifier que :

1. L'identifiant généré n'est jamais null.
2. L'identifiant fait bien 10 caractères.

Exercice 19 : Vérification d'un singleton

Crée une classe Configuration qui suit le modèle Singleton (une seule instance possible).

Écris un test JUnit pour vérifier que deux appels à Configuration.getInstance() retournent bien la même instance.

Exercice 20 : Test d'une méthode de tri

Crée une méthode trierListe(List<String> noms) qui trie une liste de noms par ordre alphabétique.

Ajoute un test unitaire pour vérifier que :

1. La liste triée est correcte.
2. Une liste vide ne pose pas de problème.
3. Une liste avec un seul élément reste inchangée.