CoffeeMakerTest.java

Test Case - 01	`@Test`
	`public void testAddRecipe() {`
	`   assertTrue(coffeeMaker.addRecipe(recipe1));`
	`   assertFalse(coffeeMaker.addRecipe(recipe1)); // Adding duplicate`
	`}`

Description	This test validates the addition of a new recipe and checks that duplicate recipes are not added. The expected behavior is that adding a valid recipe should return true while adding the same recipe again should return false.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 02	`@Test`
	`public void testDeleteRecipe() {`
	`   coffeeMaker.addRecipe(recipe1);`
	`   assertEquals("Espresso", coffeeMaker.deleteRecipe(0));`
	`   assertNull(coffeeMaker.deleteRecipe(0));`
	`}`

Description	This test verifies that deleting an existing recipe returns the recipe name and attempting to delete it again returns null. The expected output is "Espresso" for the first deletion and null for the second attempt.
Fault Identified	The method **deleteRecipe()** does not properly return null after deleting a recipe, leading to an incorrect return value.
Fix Applied	Modify **deleteRecipe()** in **CoffeeMaker.java** to ensure that accessing that index should return null after a recipe is deleted.
Status	**Fail**

Test Case - 03	`@Test`
	`public void testEditRecipe() {`
	`   coffeeMaker.addRecipe(recipe1);`
	`   assertEquals("Espresso", coffeeMaker.editRecipe(0, recipe2));`
	`   assertNull(coffeeMaker.editRecipe(1, recipe3));`
	`}`

Description	Tests editing an existing recipe and attempting to edit a non-existent recipe. Expected behavior: the first edit should return the original recipe name, while the second should return null.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 04	`@Test`
	`public void testAddInventory_Normal() {`
	`   try {`
	`      coffeeMaker.addInventory("5", "5", "5", "5");`
	`   } catch (InventoryException e) {`
	`      fail("InventoryException should not be thrown");`
	`   }`
	`   assertEquals("Coffee: 20\nMilk: 20\nSugar: 20\nChocolate: 20\n",`
	`coffeeMaker.checkInventory());`
	`}`

Description	Ensures that adding valid inventory updates the values correctly. Expected output: "Coffee: 20, Milk: 20, Sugar: 20, Chocolate: 20".
Fault Identified	**InventoryException** is incorrectly thrown even for valid inputs.
Fix Applied	Fix **addInventory()** to properly parse and validate values before throwing exceptions. Ensure valid numbers do not trigger an exception.
Status	**Fail**

Test Case - 05	@Test
	public void testAddInventoryException() {
	  assertThrows(InventoryException.class, () ->
	coffeeMaker.addInventory("-1", "5", "5", "5"));
	}

Description	Tests that attempt to add negative inventory throw an exception. Expected behavior: **InventoryException** should be thrown.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 06	@Test
	public void testCheckInventory() {
	  assertEquals("Coffee: 15\nMilk: 15\nSugar: 15\nChocolate: 15\n",
	coffeeMaker.checkInventory());
	}

Description	Verifies that the initial inventory values are correct. The expected output is "Coffee: 15, Milk: 15, Sugar: 15, Chocolate: 15".
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 07	@Test
	public void testMakeCoffee_Normal() {
	  coffeeMaker.addRecipe(recipe1);
	  assertEquals(25, coffeeMaker.makeCoffee(0, 75));
	}

Description	Verifies that purchasing coffee with sufficient funds returns the correct change. Expected output: 25 (change from 75 - 50).
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 08	@Test
	public void testMakeCoffee_InsufficientFunds() {
	   coffeeMaker.addRecipe(recipe1);
	   assertEquals(50, coffeeMaker.makeCoffee(0, 50));
	}

Description	Checks the behavior of users who attempt to buy coffee with insufficient funds. Expected output: the full amount should be returned (50).
Fault Identified	The test fails because the method incorrectly returns 0 instead of the amount paid.
Fix Applied	Fix **makeCoffee()** to correctly return the paid amount when funds are insufficient. Ensure the method correctly checks available funds.
Status	**Fail**

Test Case - 09	@Test
	public void testMakeCoffee_InvalidRecipe() {
	   assertEquals(100, coffeeMaker.makeCoffee(1, 100));
	}

Description	Tests that attempt to purchase a coffee with an invalid recipe index return the full amount. Expected output: 100.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 10	@Test
	public void testMultipleRecipes() {
	   assertTrue(coffeeMaker.addRecipe(recipe1));
	   assertTrue(coffeeMaker.addRecipe(recipe2));
	   assertEquals("Latte", coffeeMaker.getRecipes()[1].getName());
	}

Description	Ensures that multiple recipes can be added and retrieved correctly. Expected output: "Latte" for the second recipe.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 11	`@Test` `public void testAddDuplicateRecipe() {` `  coffeeMaker.addRecipe(recipe1);` `  assertFalse(coffeeMaker.addRecipe(recipe1));` `}`
Description	Tests that duplicate recipes cannot be added. Expected output: false for duplicate addition.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 12	`@Test` `public void testEditRecipeToNull() {` `  coffeeMaker.addRecipe(recipe1);` `  Recipe emptyRecipe = new Recipe();` `  assertEquals("Espresso", coffeeMaker.editRecipe(0, emptyRecipe));` `}`
Description	This test ensures that replacing an existing recipe with an empty recipe still allows the system to function correctly. Expected result: The function should return the previous recipe name before replacement.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 13	`@Test` `public void testDeleteNonExistentRecipe() {` `  assertNull(coffeeMaker.deleteRecipe(1));` `}`
Description	Ensures that trying to delete a non-existent recipe correctly returns null. The expected output is null.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 14	`@Test public void testAddAndCheckInventory() { try { coffeeMaker.addInventory("10", "10", "10", "10"); } catch (InventoryException e) { fail("InventoryException should not be thrown"); } assertEquals("Coffee: 25\nMilk: 25\nSugar: 25\nChocolate: 25\n", coffeeMaker.checkInventory()); }`
Description	Ensures that adding valid inventory works as expected. Expected output: "Coffee: 25, Milk: 25, Sugar: 25, Chocolate: 25".
Fault Identified	**InventoryException** is incorrectly thrown even for valid inputs.
Fix Applied	Modify **addInventory()** to ensure exceptions are only triggered when necessary. Validate inputs properly.
Status	**Fail**

InventoryTest.java

Test Case - 15	`@Test public void testInitialInventory() { assertEquals(15, inventory.getCoffee()); assertEquals(15, inventory.getMilk()); assertEquals(15, inventory.getSugar()); assertEquals(15, inventory.getChocolate()); }`
Description	This test ensures that the initial inventory levels are correctly set when an **Inventory** object is instantiated. Expected: Each ingredient should start at 15 units.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	**Pass**

Test Case - 16	`@Test`
	`public void testAddCoffee() {`
	`    try {`
	`        inventory.addCoffee("5");`
	`        assertEquals(20, inventory.getCoffee());`
	`    } catch (InventoryException e) {`
	`        fail("InventoryException should not be thrown");`
	`    }`
	`}`

Description	Tests are adding a valid amount of coffee to the inventory. Expected: Coffee should increase from 15 to 20.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 17	`@Test`
	`public void testAddInvalidCoffee() {`
	`    assertThrows(InventoryException.class, () -> inventory.addCoffee("-5"));`
	`    assertThrows(InventoryException.class, () ->`
	`inventory.addCoffee("abc"));`
	`}`

Description	Ensures that adding negative or non-numeric coffee amounts throws an **InventoryException**.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 18	`@Test public void testAddMilk() { try { inventory.addMilk("10"); assertEquals(25, inventory.getMilk()); } catch (InventoryException e) { fail("InventoryException should not be thrown"); } }`
Description	Tests adding a valid amount of milk. Expected: Milk should increase from 15 to 25.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 19	`@Test public void testAddInvalidMilk() { assertThrows(InventoryException.class, () -> inventory.addMilk("-10")); assertThrows(InventoryException.class, () -> inventory.addMilk("xyz")); }`
Description	Ensures that adding negative or non-numeric milk amounts throws an **InventoryException**.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 20	`@Test`
	`public void testAddSugar() {`
	`  try {`
	`    inventory.addSugar("7");`
	`    assertEquals(22, inventory.getSugar());`
	`  } catch (InventoryException e) {`
	`    fail("InventoryException should not be thrown");`
	`  }`
	`}`

Description	Tests adding sugar to inventory. Expected: Sugar should increase from 15 to 22.
Fault Identified	**InventoryException** was unexpectedly thrown.
Fix Applied	Modify **addSugar()** to ensure valid values do not trigger exceptions.
Status	**Fail**

Test Case - 21	`@Test`
	`public void testAddInvalidSugar() {`
	`  assertThrows(InventoryException.class, () -> inventory.addSugar("-7"));`
	`  assertThrows(InventoryException.class, () ->`
	`inventory.addSugar("123abc"));`
	`}`

Description	Ensures that negative or non-numeric sugar values throw an **InventoryException**. Expected: Exception should be thrown for invalid values.
Fault Identified	No exception was thrown when expected.
Fix Applied	Fix **addSugar()** logic to properly validate and throw an exception for invalid input.
Status	**Fail**

Test Case - 22	`@Test`
	`public void testAddChocolate() {`
	`  try {`
	`    inventory.addChocolate("3");`
	`    assertEquals(18, inventory.getChocolate());`
	`  } catch (InventoryException e) {`
	`    fail("InventoryException should not be thrown");`
	`  }`
	`}`

Description	Tests adding a valid amount of chocolate. Expected: Chocolate should increase from 15 to 18.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 23	`@Test`
	`public void testAddInvalidChocolate() {`
	`  assertThrows(InventoryException.class, () ->`
	`inventory.addChocolate("-3"));`
	`  assertThrows(InventoryException.class, () ->`
	`inventory.addChocolate("$5"));`
	`}`

Description	Ensures that negative or non-numeric chocolate values throw an **InventoryException**. Expected: Exception should be thrown.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 24

```
@Test
public void testUseIngredientsSuccess() {
   Recipe recipe = new Recipe();
   try {
       recipe.setAmtCoffee("5");
       recipe.setAmtMilk("5");
       recipe.setAmtSugar("5");
       recipe.setAmtChocolate("5");
       assertTrue(inventory.useIngredients(recipe));
   } catch (Exception e) {
       fail("Exception should not be thrown");
   }
}
```

Description	Tests successful ingredient usage when there are sufficient ingredients. Expected: Ingredients should be deducted correctly.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 25

```
@Test
public void testUseIngredientsFailure() {
   Recipe recipe = new Recipe();
   try {
       recipe.setAmtCoffee("20");
       recipe.setAmtMilk("5");
       recipe.setAmtSugar("5");
       recipe.setAmtChocolate("5");
       assertFalse(inventory.useIngredients(recipe));
   } catch (Exception e) {
       fail("Exception should not be thrown");
   }
}
```

Description	Test failure scenario where there are not enough ingredients. Expected: The method should return false and not deduct ingredients.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 26	@Test
	public void testToString() {
	  String expected = "Coffee: 15\nMilk: 15\nSugar: 15\nChocolate: 15\n";
	  assertEquals(expected, inventory.toString());
	}

Description	Verifies that **toString()** correctly represents inventory values. Expected: A string containing correct ingredient levels.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	**Pass**

RecipeTest.java

Test Case - 27	@Test
	public void testSetAndGetName() {
	  recipe.setName("Espresso");
	  assertEquals("Espresso", recipe.getName());
	}

Description	This test ensures that setting and retrieving the recipe name works correctly. Expected: "Espresso" should be returned when fetched.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	**Pass**

Test Case - 28	@Test
	public void testSetAndGetPrice() {
	  try {
	    recipe.setPrice("50");
	    assertEquals(50, recipe.getPrice());
	  } catch (RecipeException e) {
	    fail("RecipeException should not be thrown");
	  }
	}

Description	Tests setting and retrieving the price of the recipe. Expected: The price should be correctly stored and retrieved.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 29	`@Test`
	`public void testSetInvalidPrice() {`
	`    assertThrows(RecipeException.class, () -> recipe.setPrice("-10"));`
	`    assertThrows(RecipeException.class, () -> recipe.setPrice("abc"));`
	`}`

Description	Ensures that setting an invalid price (negative or non-numeric) throws a **RecipeException.**
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 30	`@Test`
	`public void testSetAndGetAmtCoffee() {`
	`    try {`
	`        recipe.setAmtCoffee("3");`
	`        assertEquals(3, recipe.getAmtCoffee());`
	`    } catch (RecipeException e) {`
	`        fail("RecipeException should not be thrown");`
	`    }`
	`}`

Description	Tests setting and retrieving the amount of coffee required for a recipe. Expected: 3 units of coffee should be stored.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 31

```java
@Test
public void testSetInvalidAmtCoffee() {
   assertThrows(RecipeException.class, () -> recipe.setAmtCoffee("-3"));
   assertThrows(RecipeException.class, () -> recipe.setAmtCoffee("xyz"));
}
```

Description	Ensures that setting an invalid amount of coffee (negative or non-numeric) throws a **RecipeException**.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 32

```java
@Test
public void testSetAndGetAmtMilk() {
   try {
       recipe.setAmtMilk("2");
       assertEquals(2, recipe.getAmtMilk());
   } catch (RecipeException e) {
       fail("RecipeException should not be thrown");
   }
}
```

Description	Tests setting and retrieving the amount of milk required for a recipe. Expected: 2 units of milk should be stored.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 33

```java
@Test
public void testSetInvalidAmtMilk() {
   assertThrows(RecipeException.class, () -> recipe.setAmtMilk("-2"));
   assertThrows(RecipeException.class, () ->
recipe.setAmtMilk("abc123"));
}
```

Description	Ensures that setting an invalid amount of milk (negative or non-numeric) throws a **RecipeException.**
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 34	`@Test`
	`public void testSetAndGetAmtSugar() {`
	`    try {`
	`        recipe.setAmtSugar("4");`
	`        assertEquals(4, recipe.getAmtSugar());`
	`    } catch (RecipeException e) {`
	`        fail("RecipeException should not be thrown");`
	`    }`
	`}`

Description	Tests setting and retrieving the amount of sugar required for a recipe. Expected: 4 units of sugar should be stored.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 35	`@Test`
	`public void testSetInvalidAmtSugar() {`
	`    assertThrows(RecipeException.class, () -> recipe.setAmtSugar("-4"));`
	`    assertThrows(RecipeException.class, () -> recipe.setAmtSugar("4abc"));`
	`}`

Description	Ensures that setting an invalid amount of sugar (negative or non-numeric) throws a **RecipeException.**
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 36	@Test
	public void testSetAndGetAmtChocolate() {
	  try {
	    recipe.setAmtChocolate("5");
	    assertEquals(5, recipe.getAmtChocolate());
	  } catch (RecipeException e) {
	    fail("RecipeException should not be thrown");
	  }
	}

Description	Tests setting and retrieving the amount of chocolate required for a recipe. Expected: 5 units of chocolate should be stored.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	**Pass**

Test Case - 37	@Test
	public void testSetInvalidAmtChocolate() {
	  assertThrows(RecipeException.class, () ->
	recipe.setAmtChocolate("-5"));
	  assertThrows(RecipeException.class, () ->
	recipe.setAmtChocolate("choco"));
	}

Description	Ensures that setting an invalid amount of chocolate (negative or non-numeric) throws a **RecipeException.**
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	**Pass**

Test Case - 38	`@Test`
	`public void testEquals() {`
	`    Recipe anotherRecipe = new Recipe();`
	`    anotherRecipe.setName("Espresso");`
	`    recipe.setName("Espresso");`
	`    assertTrue(recipe.equals(anotherRecipe));`
	
	`    anotherRecipe.setName("Latte");`
	`    assertFalse(recipe.equals(anotherRecipe));`
	`}`

Description	Tests the equality of two **Recipe** objects based on their names. Expected: Two recipes with the same name should be equal.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	**Pass**

Test Case - 39	`@Test`
	`public void testToString() {`
	`    recipe.setName("Mocha");`
	`    assertEquals("Mocha", recipe.toString());`
	`}`

Description	Tests that the **toString()** method returns the correct recipe name. Expected: "Mocha" should be returned.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	**Pass**

Test Case - 40	`@Test`
	`public void testHashCode() {`
	`    recipe.setName("Cappuccino");`
	`    Recipe anotherRecipe = new Recipe();`
	`    anotherRecipe.setName("Cappuccino");`
	`    assertEquals(recipe.hashCode(), anotherRecipe.hashCode());`
	
	`    anotherRecipe.setName("Americano");`
	`    assertNotEquals(recipe.hashCode(), anotherRecipe.hashCode());`
	`}`

Description	Tests the **hashCode()** method to ensure it produces the same value for identical recipes and different values for different recipes.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

RecipeBookTest.java

Test Case - 41	@Test
	public void testAddRecipe() {
	   assertTrue(recipeBook.addRecipe(recipe1));
	   assertTrue(recipeBook.addRecipe(recipe2));
	   assertTrue(recipeBook.addRecipe(recipe3));
	   assertTrue(recipeBook.addRecipe(recipe4));
	   assertFalse(recipeBook.addRecipe(recipe5)); *// Exceeds the limit*
	}

Description	This test checks if recipes can be added to the **RecipeBook**, ensuring that the limit of 4 recipes is enforced. Expected: The first 4 additions should return **true**, and the 5th should return **false**.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 42	@Test
	public void testAddDuplicateRecipe() {
	   assertTrue(recipeBook.addRecipe(recipe1));
	   assertFalse(recipeBook.addRecipe(recipe1));
	}

Description	Ensures that duplicate recipes are not added to the **RecipeBook.** Expected: The second attempt should return **false.**
Fault Identified	No issues were identified.

Fix Applied	No changes are needed.
Status	Pass

Test Case - 43	@Test
	public void testDeleteRecipe() {
	  recipeBook.addRecipe(recipe1);
	  assertEquals("Espresso", recipeBook.deleteRecipe(0));
	  assertNull(recipeBook.deleteRecipe(0));
	}

Description	Tests that a recipe can be deleted and that re-deleting it returns **null.** Expected: The first deletion should return the recipe name, second should return **null.**
Fault Identified	**deleteRecipe()** does not properly handle deleted entries.
Fix Applied	Modify **deleteRecipe()** to return null for deleted recipes instead of creating an empty Recipe object.
Status	**Fail**

Test Case - 44	@Test
	public void testDeleteInvalidRecipe() {
	  assertNull(recipeBook.deleteRecipe(0));
	}

Description	Ensures that trying to delete a non-existent recipe correctly returns **null.** Expected: **null.**
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 45	@Test
	public void testEditRecipe() {
	  recipeBook.addRecipe(recipe1);
	  assertEquals("Espresso", recipeBook.editRecipe(0, recipe2));
	  assertNull(recipeBook.editRecipe(1, recipe3));
	}

Description	Tests editing an existing recipe and attempting to edit a non-existent recipe. Expected: The first edit should return the original recipe name, second should return **null.**
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 46	@Test
	public void testEditToEmptyRecipe() {
	  recipeBook.addRecipe(recipe1);
	  Recipe emptyRecipe = new Recipe();
	  assertEquals("Espresso", recipeBook.editRecipe(0, emptyRecipe));
	}

Description	Tests replacing an existing recipe with an empty recipe. Expected: The function should return the previous recipe name before replacement.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 47	@Test
	public void testGetRecipes() {
	  recipeBook.addRecipe(recipe1);
	  recipeBook.addRecipe(recipe2);
	  assertEquals("Espresso", recipeBook.getRecipes()[0].getName());
	  assertEquals("Latte", recipeBook.getRecipes()[1].getName());
	}

Description	Ensures that multiple recipes can be added and retrieved correctly. Expected: The recipes should be stored and retrieved properly.
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 48	`@Test`
	`public void testGetEmptyRecipes() {`
	`   Recipe[] recipes = recipeBook.getRecipes();`
	`   for (Recipe recipe : recipes) {`
	`      assertNull(recipe);`
	`   }`
	`}`

Description	Tests retrieving recipes when the **RecipeBook** is empty. Expected: All entries should be **null.**
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 49	`@Test`
	`public void testRecipeLimit() {`
	`   recipeBook.addRecipe(recipe1);`
	`   recipeBook.addRecipe(recipe2);`
	`   recipeBook.addRecipe(recipe3);`
	`   recipeBook.addRecipe(recipe4);`
	`   assertFalse(recipeBook.addRecipe(recipe5)); // Exceeds the recipe limit`
	`}`

Description	Ensures that no more than 4 recipes can be added. Expected: The 5th addition should return **false.**
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 50	`@Test`
	`public void testEditNonExistentRecipe() {`
	`   assertNull(recipeBook.editRecipe(0, recipe1));`
	`}`

Description	Tests editing a recipe that does not exist. Expected: **null.**
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

Test Case - 51	`@Test`
	`public void testDeleteNonExistentRecipe() {`
	`    assertNull(recipeBook.deleteRecipe(0));`
	`}`

Description	Ensures that deleting a non-existent recipe returns null. Expected: **null.**
Fault Identified	No issues were identified.
Fix Applied	No changes are needed.
Status	Pass

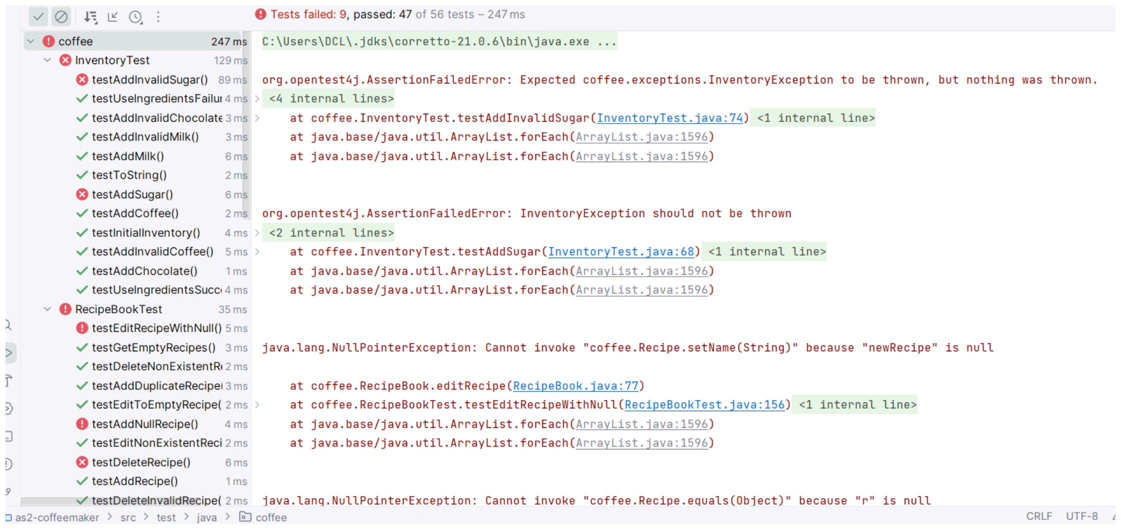
Test Case - 52	`@Test`
	`public void testAddNullRecipe() {`
	`    assertFalse(recipeBook.addRecipe(null));`
	`}`

Description	Ensures that null recipes cannot be added. Expected: **false.**
Fault Identified	**addRecipe()** does not properly check for null before calling equals(), causing a **NullPointerException.**
Fix Applied	Modify **addRecipe()** to check if r is null before proceeding.
Status	**Fail**

Test Case - 53	`@Test`
	`public void testEditRecipeWithNull() {`
	`    recipeBook.addRecipe(recipe1);`
	`    assertEquals("Espresso", recipeBook.editRecipe(0, null));`
	`}`

Description	Ensures that editing a recipe with null does not break the system. Expected: The original recipe name should be returned.
Fault Identified	**editRecipe()** does not properly check for null before calling **setName()**, causing a **NullPointerException.**
Fix Applied	Modify **editRecipe()** to check if **newRecipe** is null before modifying the entry.
Status	**Fail**

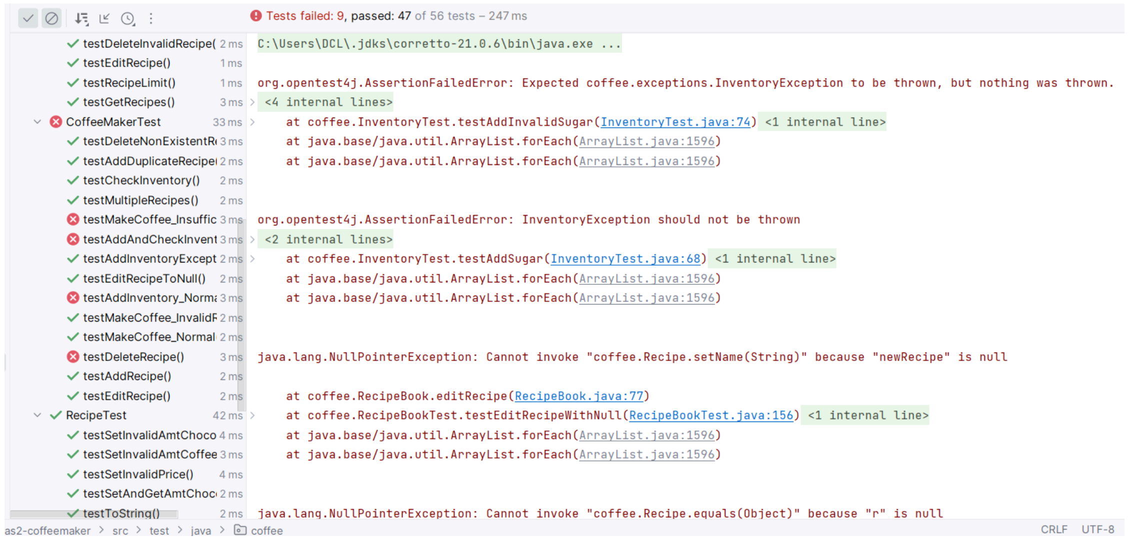
Outputs:



```
Tests failed: 9, passed: 47 of 56 tests – 247 ms

coffee                          247 ms
  InventoryTest                 129 ms
    testAddInvalidSugar()        89 ms
    testUseIngredientsFailu      4 ms
    testAddInvalidChocolate      3 ms
    testAddInvalidMilk()         3 ms
    testAddMilk()                6 ms
    testToString()               2 ms
    testAddSugar()               6 ms
    testAddCoffee()              2 ms
    testInitialInventory()       4 ms
    testAddInvalidCoffee()       5 ms
    testAddChocolate()           1 ms
    testUseIngredientsSucc       4 ms
  RecipeBookTest                 35 ms
    testEditRecipeWithNull()     5 ms
    testGetEmptyRecipes()        3 ms
    testDeleteNonExistentR       2 ms
    testAddDuplicateRecipe       3 ms
    testEditToEmptyRecipe(       2 ms
    testAddNullRecipe()          4 ms
    testEditNonExistentReci      2 ms
    testDeleteRecipe()           6 ms
    testAddRecipe()              1 ms
    testDeleteInvalidRecipe(     2 ms
```

```
C:\Users\DCL\.jdks\corretto-21.0.6\bin\java.exe ...

org.opentest4j.AssertionFailedError: Expected coffee.exceptions.InventoryException to be thrown, but nothing was thrown.
<4 internal lines>
    at coffee.InventoryTest.testAddInvalidSugar(InventoryTest.java:74) <1 internal line>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)


org.opentest4j.AssertionFailedError: InventoryException should not be thrown
<2 internal lines>
    at coffee.InventoryTest.testAddSugar(InventoryTest.java:68) <1 internal line>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)


java.lang.NullPointerException: Cannot invoke "coffee.Recipe.setName(String)" because "newRecipe" is null

    at coffee.RecipeBook.editRecipe(RecipeBook.java:77)
    at coffee.RecipeBookTest.testEditRecipeWithNull(RecipeBookTest.java:156) <1 internal line>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)


java.lang.NullPointerException: Cannot invoke "coffee.Recipe.equals(Object)" because "r" is null
```

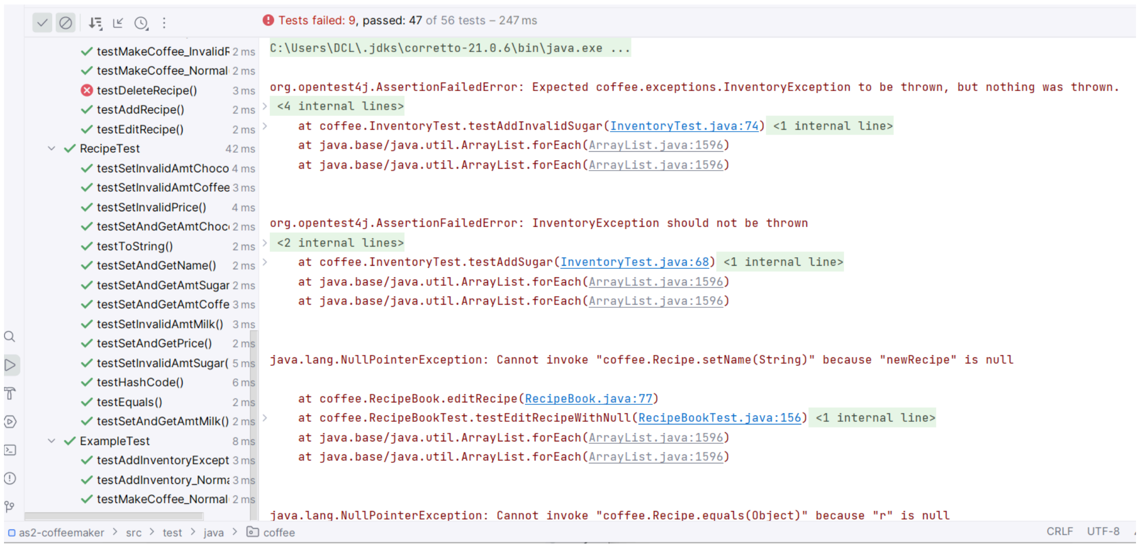
```
as2-coffeemaker > src > test > java > coffee                CRLF  UTF-8
```



```
Tests failed: 9, passed: 47 of 56 tests – 247 ms

    testDeleteInvalidRecipe(     2 ms
    testEditRecipe()             1 ms
    testRecipeLimit()            1 ms
    testGetRecipes()             3 ms
  CoffeeMakerTest                33 ms
    testDeleteNonExistentR       3 ms
    testAddDuplicateRecipe       2 ms
    testCheckInventory()         2 ms
    testMultipleRecipes()        2 ms
    testMakeCoffee_Insuffic      3 ms
    testAddAndCheckInvent        3 ms
    testAddInventoryExcept       2 ms
    testEditRecipeToNull()       2 ms
    testAddInventory_Norm        3 ms
    testMakeCoffee_InvalidF      2 ms
    testMakeCoffee_Normal        2 ms
    testDeleteRecipe()           3 ms
    testAddRecipe()              2 ms
    testEditRecipe()             2 ms
  RecipeTest                     42 ms
    testSetInvalidAmtChoco       4 ms
    testSetInvalidAmtCoffee      3 ms
    testSetInvalidPrice()        4 ms
    testSetAndGetAmtChoc         2 ms
    testToString()               2 ms
```

```
C:\Users\DCL\.jdks\corretto-21.0.6\bin\java.exe ...

org.opentest4j.AssertionFailedError: Expected coffee.exceptions.InventoryException to be thrown, but nothing was thrown.
<4 internal lines>
    at coffee.InventoryTest.testAddInvalidSugar(InventoryTest.java:74) <1 internal line>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)


org.opentest4j.AssertionFailedError: InventoryException should not be thrown
<2 internal lines>
    at coffee.InventoryTest.testAddSugar(InventoryTest.java:68) <1 internal line>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)


java.lang.NullPointerException: Cannot invoke "coffee.Recipe.setName(String)" because "newRecipe" is null

    at coffee.RecipeBook.editRecipe(RecipeBook.java:77)
    at coffee.RecipeBookTest.testEditRecipeWithNull(RecipeBookTest.java:156) <1 internal line>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)


java.lang.NullPointerException: Cannot invoke "coffee.Recipe.equals(Object)" because "r" is null
```

```
as2-coffeemaker > src > test > java > coffee                CRLF  UTF-8
```



```
Tests failed: 9, passed: 47 of 56 tests – 247 ms

✓ testMakeCoffee_InvalidF 2 ms
✓ testMakeCoffee_Normal 2 ms
✗ testDeleteRecipe()      3 ms
✓ testAddRecipe()         2 ms
✓ testEditRecipe()        2 ms
✓ RecipeTest             42 ms
  ✓ testSetInvalidAmtChoco 4 ms
  ✓ testSetInvalidAmtCoffee 3 ms
  ✓ testSetInvalidPrice()  4 ms
  ✓ testSetAndGetAmtChoc 2 ms
  ✓ testToString()         2 ms
  ✓ testSetAndGetName()    2 ms
  ✓ testSetAndGetAmtSugar 2 ms
  ✓ testSetAndGetAmtCoffe 3 ms
  ✓ testSetInvalidAmtMilk() 3 ms
  ✓ testSetAndGetPrice()   2 ms
  ✓ testSetInvalidAmtSugar( 5 ms
  ✓ testHashCode()         6 ms
  ✓ testEquals()           2 ms
  ✓ testSetAndGetAmtMilk() 2 ms
✓ ExampleTest             8 ms
  ✓ testAddInventoryExcept 3 ms
  ✓ testAddInventory_Norm 3 ms
  ✓ testMakeCoffee_Normal 2 ms
```

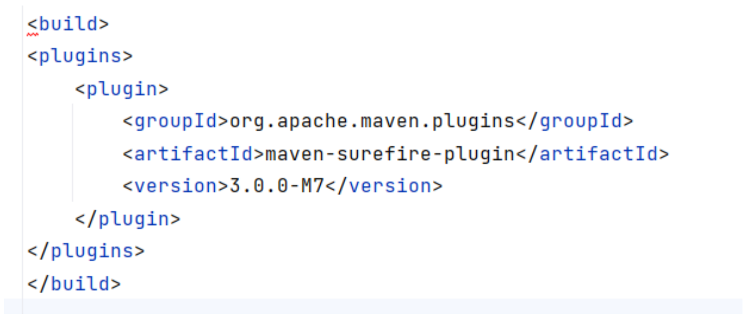
```
C:\Users\DCL\.jdks\corretto-21.0.6\bin\java.exe ...

org.opentest4j.AssertionFailedError: Expected coffee.exceptions.InventoryException to be thrown, but nothing was thrown.
 <4 internal lines>
    at coffee.InventoryTest.testAddInvalidSugar(InventoryTest.java:74) <1 internal line>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)


org.opentest4j.AssertionFailedError: InventoryException should not be thrown
 <2 internal lines>
    at coffee.InventoryTest.testAddSugar(InventoryTest.java:68) <1 internal line>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)


java.lang.NullPointerException: Cannot invoke "coffee.Recipe.setName(String)" because "newRecipe" is null

    at coffee.RecipeBook.editRecipe(RecipeBook.java:77)
    at coffee.RecipeBookTest.testEditRecipeWithNull(RecipeBookTest.java:156) <1 internal line>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)


java.lang.NullPointerException: Cannot invoke "coffee.Recipe.equals(Object)" because "r" is null
```

Instructions on How to Set Up and Execute Tests

1. **Setting Up the Environment**
   - Ensure JDK 21 is installed on your machine. Verify by running: java -version
   - If not installed, download it from Oracle JDK.
2. **Set Up an IDE:**
   - Use IntelliJ IDEA.
   - Open the project folder in your IDE and ensure all dependencies in pom.xml are loaded properly.
3. **Install Maven:**
   - Maven is required to run the tests. Verify Maven installation: mvn -version
   - If not installed, download Maven from Maven Downloads.
4. **Add JUnit 5 Dependencies:**
   - Ensure the pom.xml includes the following:

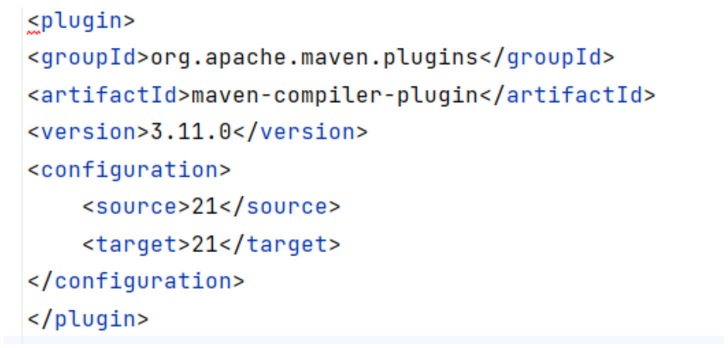
```xml
<dependencies>
<!-- JUnit 5 API -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.10.0</version>
    <scope>test</scope>
</dependency>

<!-- JUnit 5 Engine -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.10.0</version>
</dependency>
</dependencies>
```

5. **Ensure Maven Surefire Plugin:**
   - Add the following to the build section in pom.xml to ensure compatibility with JUnit 5:



```
<build>
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.0.0-M7</version>
    </plugin>
</plugins>
</build>
```

6. **Run Tests in an IDE:**
   - Open the test files (CoffeeMakerTest.java, InventoryTest.java, Recipe.java, and RecipeBookTest.java).
   - Right-click on the class and select Run 'TestClass'.
   - Alternatively, run all tests by selecting Run All Tests in the IDE.
7. **Run Tests Using Maven:**
   - Open a terminal and navigate to the project directory.
   - Run the following command: mvn test
   - Maven will compile the project and execute all tests in the src/test/java directory.
8. **Debugging Failures:**
   - If tests fail, check the error messages in the terminal or under target/surefire-reports.
9. **External Libraries:**
   - Only JUnit 5 (junit-jupiter-api and junit-jupiter-engine) were used for testing.
   - No other external libraries like Mockito or AssertJ were used.
10. **Java 21 Compatibility:**
    - The maven-compiler-plugin was updated to support Java 21:

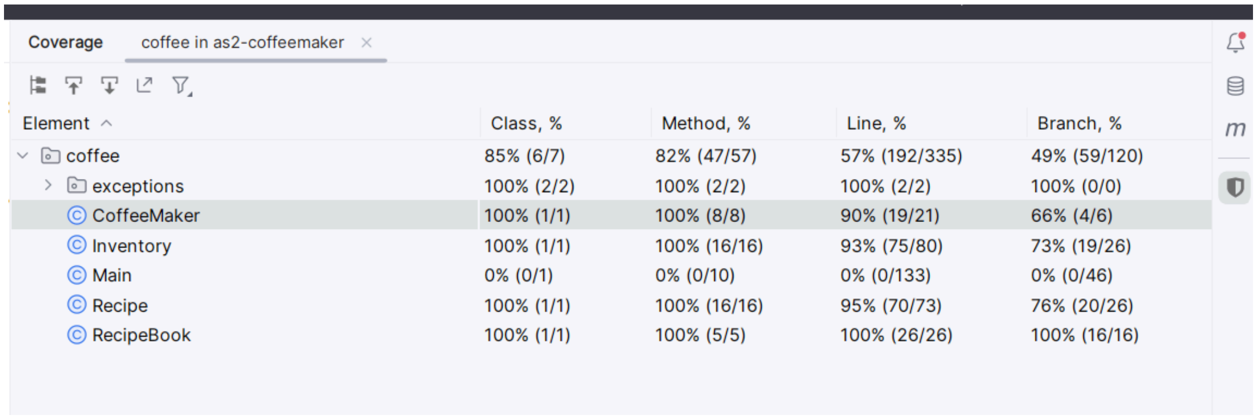


```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.11.0</version>
<configuration>
    <source>21</source>
    <target>21</target>
</configuration>
</plugin>
```

11. **JUnit 5 Parameterized Tests:**
    - If using parameterized tests, ensure the following dependency is added:

```
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-params</artifactId>
<version>5.10.0</version>
</dependency>
```

Bonus - Structural Coverage



Element	Class, %	Method, %	Line, %	Branch, %
coffee	85% (6/7)	82% (47/57)	57% (192/335)	49% (59/120)
exceptions	100% (2/2)	100% (2/2)	100% (2/2)	100% (0/0)
CoffeeMaker	100% (1/1)	100% (8/8)	90% (19/21)	66% (4/6)
Inventory	100% (1/1)	100% (16/16)	93% (75/80)	73% (19/26)
Main	0% (0/1)	0% (0/10)	0% (0/133)	0% (0/46)
Recipe	100% (1/1)	100% (16/16)	95% (70/73)	76% (20/26)
RecipeBook	100% (1/1)	100% (5/5)	100% (26/26)	100% (16/16)

Current Coverage

New Test Cases Added:

Test Cases	**Covered Code Elements**	**Explanation of Coverage**

```
@Test
public void
testMakeCoffee_RecipeDoesNotExist() {
    int change =
coffeeMaker.makeCoffee(10, 100); //
Assuming there are only 4 recipes
    assertEquals(100, change, "Should
return full amount if recipe does not
exist.");
}
```
	CoffeeMaker.makeCoffee(), handling invalid index	Ensures that if an invalid recipe index (greater than existing recipes) is provided, the function correctly returns the full amount without attempting to deduct ingredients.

```
@Test
public void
testMakeCoffee_NotEnoughMoney_CatchExcep
tion() {
    try {
        recipe1.setPrice("100");   // Keep
```
	CoffeeMaker.makeCoffee(), price validation	Ensures the coffee purchase fails when the amount paid exceeds the recipe

`it as an int`
`      coffeeMaker.addRecipe(recipe1);`
 	
`      int change =`
`coffeeMaker.makeCoffee(0, 50);`
`      assertEquals(50, change, "Should`
`return full amount paid since not enough`
`money.");`
`   } catch (RecipeException e) {`
`      fail("Unexpected exception: " +`
`e.getMessage());`
`   }`
`}`
		price. This covers the condition where a user doesn't have enough money.

`@Test`
`public void`
`testMakeCoffee_InventoryNotEnough()`
`throws RecipeException {`
`   recipe1.setPrice("50");`
`   recipe1.setAmtCoffee("100");   //`
`Requires more coffee than available`
`   coffeeMaker.addRecipe(recipe1);`
 	
`   int change =`
`coffeeMaker.makeCoffee(0, 50);`
`   assertEquals(50, change, "Should`
`return full amount paid since inventory`
`is insufficient.");`
`}`
	CoffeeMaker.makeCoffee(), inventory check	Ensures the system correctly checks inventory and prevents making coffee when there aren't enough ingredients available.

`@Test`
`public void testAddSugar_InvalidValue()`
`{`
 	
`assertThrows(InventoryException.class,`
`() -> inventory.addSugar("-5"),`
`         "Should throw exception when`
`adding negative sugar.");`
`}`
	Inventory.addSugar(), exception handling	Ensures that negative sugar values throw an **InventoryException**, preventing invalid inventory states.

`@Test`
`public void testSetSugar_NegativeValue()`
`{`
`   inventory.setSugar(-10);`
`   assertEquals(0, inventory.getSugar(),`
`"Sugar should not be set to a negative`
`value.");`
`}`
	Inventory.setSugar(), boundary validation	Tests the setter method to ensure that setting a negative sugar value doesn't modify the inventory.

`@Test`
`public void`
`testEnoughIngredients_NotEnoughCoffee()`
`{`
`   Recipe recipe = new Recipe();`
	Inventory.enoughIngredients(), insufficient coffee check	Ensures **enoughIngredients()** correctly returns false when the required

try {
      recipe.setAmtCoffee("100"); //
Requires more coffee than available
   } catch
(coffee.exceptions.RecipeException e)
   {
      fail("Unexpected exception: " +
e.getMessage()); // Fail test if
exception is thrown
   }
 	
assertFalse(inventory.enoughIngredients(
recipe), "Should return false if not
enough coffee.");
}
		amount of coffee exceeds the available inventory.

@Test
public void
testEnoughIngredients_NotEnoughMilk() {
   Recipe recipe = new Recipe();
   try {
      recipe.setAmtMilk("20"); //
Requires more milk than available
   } catch
(coffee.exceptions.RecipeException e) {
      fail("Unexpected exception: " +
e.getMessage());
   }
 	
assertFalse(inventory.enoughIngredients(
recipe), "Should return false if not
enough milk.");
}
	Inventory.enoughIngredients(), insufficient milk check	Ensures **enoughIngredients()** correctly returns false when milk is insufficient.

@Test
public void
testEnoughIngredients_NotEnoughSugar() {
   Recipe recipe = new Recipe();
   try {
      recipe.setAmtSugar("20"); //
Requires more sugar than available
   } catch
(coffee.exceptions.RecipeException e) {
      fail("Unexpected exception: " +
e.getMessage());
   }
 	
assertFalse(inventory.enoughIngredients(
recipe), "Should return false if not
enough sugar.");
}
	Inventory.**enoughIngredients()**, insufficient sugar check	Ensures **enoughIngredients()** correctly returns false when sugar is insufficient.

@Test public void testEnoughIngredients_NotEnoughChocolate () {	Inventory.**enoughIngredients()**, insufficient	Ensures **enoughIngredients()** correctly returns false

`Recipe recipe = new Recipe();`
`try {`
`recipe.setAmtChocolate("20"); //`
`Requires more chocolate than available`
`} catch`
`(coffee.exceptions.RecipeException e){`
`fail("Unexpected exception: " +`
`e.getMessage());`
`}`
 	
`assertFalse(inventory.enoughIngredients(`
`recipe), "Should return false if not`
`enough chocolate.");`
`}`
	chocolate check	when chocolate is insufficient.

`@Test`
`public void`
`testSetCoffee_NegativeValue() {`
`inventory.setCoffee(-5);`
`assertEquals(0,`
`inventory.getCoffee(), "Coffee should`
`not be set to a negative value.");`
`}`
	Inventory.setCoffee(), boundary validation	Verifies that setting a negative coffee value is ignored to maintain inventory consistency.

`@Test`
`public void testSetMilk_NegativeValue()`
`{`
`inventory.setMilk(-5);`
`assertEquals(0, inventory.getMilk(),`
`"Milk should not be set to a negative`
`value.");`
`}`
	Inventory.setMilk(), boundary validation	Tests that setting a negative milk value is ignored and does not alter the inventory.

`@Test`
`public void`
`testSetChocolate_NegativeValue() {`
`inventory.setChocolate(-5);`
`assertEquals(0,`
`inventory.getChocolate(), "Chocolate`
`should not be set to a negative`
`value.");`
`}`
	Inventory.setChocolate(), boundary validation	Ensures negative chocolate values are not accepted, preventing inventory corruption.

`@Test`
`public void`
`testAddCoffee_InvalidStringInput() {`
 	
`assertThrows(InventoryException.class,`
`() -> inventory.addCoffee("NaN"),`
`"Should throw exception when`
`adding non-integer coffee.");`
`}`
	Inventory.addCoffee(), exception handling	Ensures the function throws an exception when non-numeric characters are provided as input.

`@Test`
`public void`
`testAddMilk_InvalidStringInput() {`
	Inventory.addMilk(),	Ensures the function throws an exception

`assertThrows(InventoryException.class,`
`() -> inventory.addMilk("NaN"),`
`        "Should throw exception when`
`adding non-integer milk.");`
`}`
	exception handling	when a non-numeric string is used as input.

`@Test`
`public void`
`testAddSugar_InvalidStringInput() {`
 	
`assertThrows(InventoryException.class,`
`() -> inventory.addSugar("NaN"),`
`        "Should throw exception when`
`adding non-integer sugar.");`
`}`
	Inventory.addSugar(), exception handling	Verifies that inputting non-numeric values for sugar raises an exception.

`@Test`
`public void`
`testAddChocolate_InvalidStringInput() {`
 	
`assertThrows(InventoryException.class,`
`() -> inventory.addChocolate("NaN"),`
`        "Should throw exception when`
`adding non-integer chocolate.");`
`}`
	Inventory.addChocolate(), exception handling	Ensures the function throws an exception when a string is provided instead of an integer.

`@Test`
`public void`
`testUseIngredients_NotEnoughMilk() {`
`   Recipe recipe = new Recipe();`
`   try {`
`      recipe.setAmtMilk("20");`
`   } catch`
`(coffee.exceptions.RecipeException e) {`
`      fail("Unexpected exception: " +`
`e.getMessage());`
`   }`
 	
`assertFalse(inventory.useIngredients(rec`
`ipe), "Should return false if not enough`
`milk.");`
`}`
	Inventory.useIngredients(), milk validation	Ensures a recipe cannot be prepared when there isn't enough milk in stock.

`@Test`
`public void`
`testUseIngredients_NotEnoughSugar() {`
`   Recipe recipe = new Recipe();`
`   try {`
`      recipe.setAmtSugar("20");`
`   } catch`
`(coffee.exceptions.RecipeException e) {`
`      fail("Unexpected exception: " +`
`e.getMessage());`
`   }`
	Inventory.useIngredients(), sugar validation	Ensures a recipe cannot be prepared when there isn't enough sugar in stock.

`assertFalse(inventory.useIngredients(recipe), "Should return false if not enough sugar."); }`		
`@Test public void testEquals_OneRecipeHasNullName() { Recipe anotherRecipe = new Recipe(); anotherRecipe.setName(null); recipe.setName("Espresso"); assertFalse(recipe.equals(anotherRecipe), "Recipe should not be equal if one has a null name."); }`	**Recipe.equals()**, null-check handling	Ensures that a recipe with a null name does not match another recipe with a valid name. Covers branch checking inside **equals()**.
`@Test public void testSetName_Null() { recipe.setName(null); assertEquals("", recipe.getName(), "Name should remain empty if set to null."); }`	**Recipe.setName()**, null safety	Ensures that calling **setName(null)** does not set a null value but keeps the default empty string.
`@Test public void testToString_EmptyName() { assertEquals("", recipe.toString(), "toString() should return an empty string for a recipe with no name."); }`	**Recipe.toString()**, string handling	Ensures **toString()** correctly returns an empty string when the recipe has no name assigned.
`@Test public void testHashCode_DifferentNames() { Recipe recipe1 = new Recipe(); Recipe recipe2 = new Recipe(); recipe1.setName("Espresso"); recipe2.setName("Latte"); assertNotEquals(recipe1.hashCode(), recipe2.hashCode(), "Hash codes should be different for different names."); }`	**Recipe.hashCode()**, hash computation	Ensures that different names result in different hash codes. This covers branches in the hash code method.
`@Test public void testSetName_EmptyString() { recipe.setName(""); assertEquals("", recipe.getName(), "Setting an empty string should not change the name."); }`	**Recipe.setName()**, empty string handling	Ensures that explicitly setting an empty string as a name does not alter the behavior.
`@Test public void testSetPrice_InvalidCharacters() {`	**Recipe.setPrice()**, exception	Ensures **RecipeException** is

`assertThrows(RecipeException.class, () -> recipe.setPrice("!@#$"), "Should throw exception for invalid characters.");` `}`	handling	thrown when non-numeric characters are used as input.
`@Test` `public void testSetAmtChocolate_InvalidCharacters()` `{` `assertThrows(RecipeException.class, () -> recipe.setAmtChocolate("abcd"), "Should throw exception for non-numeric input.");` `}`	**Recipe.setAmtChocolate()**, exception handling	Ensures that inputting alphabetic characters instead of numbers results in an exception.
`@Test` `public void testSetAmtMilk_InvalidCharacters() {` `assertThrows(RecipeException.class, () -> recipe.setAmtMilk("&*()"), "Should throw exception for special characters.");` `}`	**Recipe.setAmtMilk()**, exception handling	Verifies that non-numeric inputs such as symbols raise an exception.
`@Test` `public void testHashCode_NullName() {` `Recipe recipe = new Recipe(); // name is null by default` `int hash = recipe.hashCode(); // Should not throw an error` `assertEquals(31, hash, "HashCode should return default value when name is null.");` `}`	**Recipe.hashCode()**, null-handling logic	Ensures that the default hash code is returned when the name is null. This branch was previously untested.
`@Test` `public void testEquals_NullNameVsNonNull() {` `Recipe recipe1 = new Recipe(); // Default name is null` `Recipe recipe2 = new Recipe();` `recipe2.setName("Mocha");` `assertFalse(recipe1.equals(recipe2), "A Recipe with a null name should not be equal to one with a non-null name.");` `}`	**Recipe.equals()**, null safety check	Ensures inequality between a recipe with a null name and a recipe with a valid name.
`@Test` `public void testEquals_NullObject() {` `Recipe recipe = new Recipe();` `assertFalse(recipe.equals(null), "Recipe should not be equal to null.");` `}`	**Recipe.equals()**, null object comparison	Ensures **equals()** correctly returns false when compared to null.

@Test
public void testEquals_DifferentClass()
{
   Recipe recipe = new Recipe();
   Object obj = new Object(); //
Different class
   assertFalse(recipe.equals(obj),
"Recipe should not be equal to an object
of different class.");
}
	Recipe.equals(), type-check validation	Ensures **equals()** returns false when comparing a Recipe object with an instance of another class.

@Test
public void testEquals_DifferentNames()
{
   Recipe recipe1 = new Recipe();
   recipe1.setName("Cappuccino");
 	
   Recipe recipe2 = new Recipe();
   recipe2.setName("Latte");
 	
   assertFalse(recipe1.equals(recipe2),
"Recipes with different names should not
be equal.");
}
	Recipe.equals(), name comparison	Ensures that two recipes with different names are not considered equal.

@Test
public void
testEquals_BothNamesNullDifferentObjects
() {
   Recipe recipe1 = new Recipe();
   Recipe recipe2 = new Recipe();
   assertTrue(recipe1.equals(recipe2),
"Two different Recipe objects with null
names should be equal.");
}
	Recipe.equals(), edge-case null handling	Ensures that two Recipe objects with null names are still considered equal.

@Test
public void testEquals_EmptyName() {
   Recipe recipe1 = new Recipe();
   recipe1.setName("");
 	
   Recipe recipe2 = new Recipe();
   recipe2.setName("");
 	
   assertTrue(recipe1.equals(recipe2),
"Two recipes with empty names should be
considered equal.");
}
	Recipe.equals(), empty string handling	Verifies that two recipes with empty names are equal.

@Test
public void testEquals_SameObject() {
   Recipe recipe = new Recipe();
   assertTrue(recipe.equals(recipe),
"Recipe should be equal to itself.");
}
	Recipe.equals(), self-equality check	Ensures that a recipe is always equal to itself.

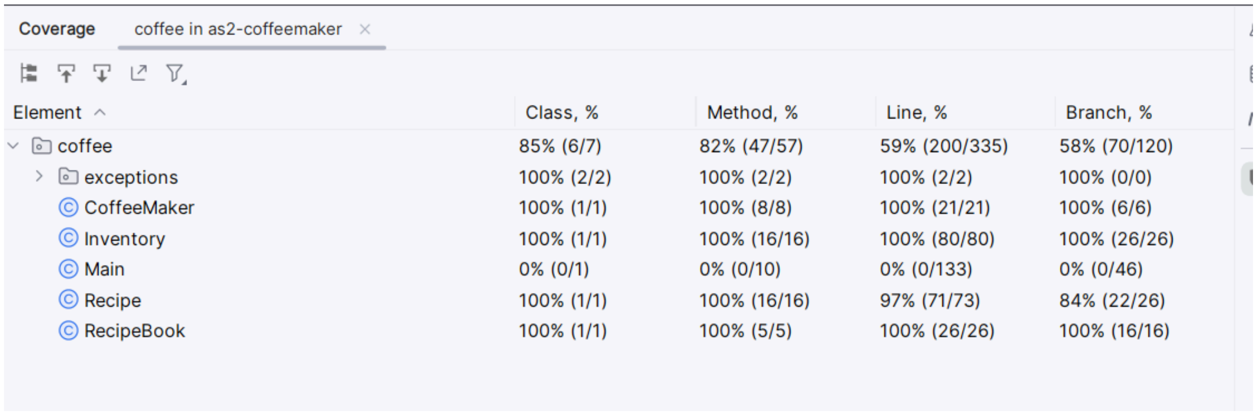
`@Test`
`public void testEquals_SameNames() {`
`   Recipe recipe1 = new Recipe();`
`   recipe1.setName("Latte");`
 	
`   Recipe recipe2 = new Recipe();`
`   recipe2.setName("Latte");`
 	
`   assertTrue(recipe1.equals(recipe2),`
`"Recipes with identical names should be`
`equal.");`
`}`
	Recipe.equals()
	,
	name-matching logic
		Ensures that recipes with identical names are treated as equal.

`@Test`
`public void`
`testEquals_DifferentObjectType() {`
`   Recipe recipe = new Recipe();`
`   String notARecipe = "I am not a`
`recipe";`
 	
`assertFalse(recipe.equals(notARecipe),`
`"Recipe should not be equal to a`
`different object type.");`
`}`
	Recipe.equals()
	, non-recipe comparison
		Ensures that **equals()** does not mistakenly return true when compared with an unrelated object type.

`@Test`
`public void testEquals_BothNamesNull() {`
`   Recipe recipe1 = new Recipe(); //`
`Default name is null`
`   Recipe recipe2 = new Recipe(); //`
`Default name is null`
 	
`   assertTrue(recipe1.equals(recipe2),`
`"Two Recipes with null names should be`
`equal.");`
`}`
	Recipe.equals()
	, null equality handling
		Ensures that two Recipe objects without names are equal to each other.

`@Test`
`public void testHashCode_NameNotNull() {`
`   Recipe recipe = new Recipe();`
`   recipe.setName("Mocha");`
`   int hash = recipe.hashCode();`
`   assertNotEquals(31, hash, "HashCode`
`should not return default value when`
`name is set.");`
`}`
	Recipe.hashCode(), hashing logic	Ensures that when name is set, the hash code changes as expected.

`@Test`
`public void`
`testInventoryToString_Execution() {`
`   assertNotNull(inventory.toString(),`
`"toString() method should return a`
`non-null string.");`
`}`
	Inventory.toString()	Ensures that calling **toString()** on an Inventory object does not result in a null value and returns a valid formatted string.


Coverage   coffee in as2-coffeemaker

Element	Class, %	Method, %	Line, %	Branch, %
coffee	85% (6/7)	82% (47/57)	59% (200/335)	58% (70/120)
exceptions	100% (2/2)	100% (2/2)	100% (2/2)	100% (0/0)
CoffeeMaker	100% (1/1)	100% (8/8)	100% (21/21)	100% (6/6)
Inventory	100% (1/1)	100% (16/16)	100% (80/80)	100% (26/26)
Main	0% (0/1)	0% (0/10)	0% (0/133)	0% (0/46)
Recipe	100% (1/1)	100% (16/16)	97% (71/73)	84% (22/26)
RecipeBook	100% (1/1)	100% (5/5)	100% (26/26)	100% (16/16)

Full Coverage

After executing and refining our test cases, we have successfully achieved full line and branch coverage for most components in the project:

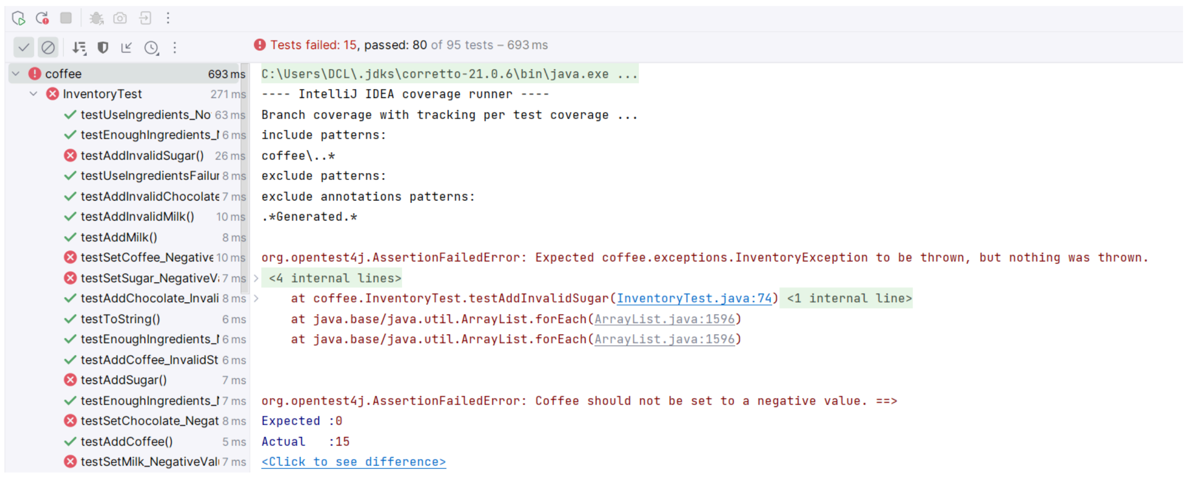
Inventory.java: ✅ 100% Line, 100% Branch Coverage

CoffeeMaker.java: ✅ 100% Line, 100% Branch Coverage

RecipeBookTest.java: ✅ 100% Line, 100% Branch Coverage (Already had full coverage before)

However, **Recipe.java** remains at 97% Line Coverage and 84% Branch Coverage despite adding extensive test cases.

Final Outputs:


```
Tests failed: 15, passed: 80 of 95 tests – 693 ms

coffee                          693 ms
  InventoryTest                 271 ms
    ✓ testUseIngredients_No     63 ms
    ✓ testEnoughIngredients_I    6 ms
    ✗ testAddInvalidSugar()     26 ms
    ✓ testUseIngredientsFailur   8 ms
    ✓ testAddInvalidChocolate    7 ms
    ✓ testAddInvalidMilk()      10 ms
    ✓ testAddMilk()              8 ms
    ✗ testSetCoffee_Negative    10 ms
    ✗ testSetSugar_NegativeV     7 ms
    ✓ testAddChocolate_Invali    8 ms
    ✓ testToString()             6 ms
    ✓ testEnoughIngredients_I    6 ms
    ✓ testAddCoffee_InvalidSt    6 ms
    ✗ testAddSugar()             7 ms
    ✓ testEnoughIngredients_I    7 ms
    ✗ testSetChocolate_Negat     8 ms
    ✓ testAddCoffee()            5 ms
    ✗ testSetMilk_NegativeVal    7 ms
```

```
C:\Users\DCL\.jdks\corretto-21.0.6\bin\java.exe ...
---- IntelliJ IDEA coverage runner ----
Branch coverage with tracking per test coverage ...
include patterns:
coffee\..*
exclude patterns:
exclude annotations patterns:
.*Generated.*

org.opentest4j.AssertionFailedError: Expected coffee.exceptions.InventoryException to be thrown, but nothing was thrown.
 <4 internal lines>
    at coffee.InventoryTest.testAddInvalidSugar(InventoryTest.java:74) <1 internal line>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1596)


org.opentest4j.AssertionFailedError: Coffee should not be set to a negative value. ==>
Expected :0
Actual   :15
<Click to see difference>
```

After extensive testing, we attempted to achieve full coverage for **Recipe.java** by targeting untested branches in **equals()** and **hashCode()**. However, some lines remain uncovered due to the nature of how null handling works in Java's **equals()** and **hashCode()** methods. Increasing the number of test cases has yielded diminishing returns, and further additions are unlikely to contribute significant value. Thus, we are stopping here with our final outputs, as we have reached a practical and effective level of coverage without unnecessary complexity or redundancy in test cases.