

NATWEST BOOTCAMP

Group Report: Group 2 Hackathon 1 & Hackathon 2

Report HACKATHON 1: 28th July, 2025

1. **TASK: Write a Junit test for Bill and Payment both for setters, date parsing with different modes of edge cases like zero amount etc**

```
1 package com.paypilot.test;
2
3 import static org.junit.jupiter.api.Assertions.*;
12
13 class PaymentSetterTest {
14     private Payment payment;
15
16     @BeforeEach
17     void setUp() {
18         payment = new Payment(1, 1, 1, LocalDate.now(), "cash");
19     }
20
21
24 void testSetPaymentId() {}
30
33 void testSetBillId() {}
38
41 void testSetAmountPaid() {}
46
49 void testSetNegativeAmountPaid() {}
56
59 void testSetRandomPaymentDate() {}
69
72 void testSetFuturePaymentDate() {}
83
86 void testSetZeroPaymentDate() {}
91
94 void testSetLeapYearPaymentDate() {}
99
102 void testSetMode() {}
107
108 }
109
```

```

1 package com.paypilot.test;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 class BillSetterTest {
6
7     private Bill bill;
8
9     @BeforeEach
10    void setUp() {
11        //create new bill object
12        bill=new Bill(0, 0, "a", "a", LocalDate.now(), 0, false);
13    }
14
15    void testSetUserId() {}
16
17    void testSetBillId() {}
18
19    void testSetBillName() {}
20
21    void testSetCategory() {}
22
23    void testSetRandomDueDate() {}
24
25    void testSetFutureDueDate() {}
26
27    void testSetZeroDueDate() {}
28
29    void testSetLeapYearDueDate() {}
30
31    void testSetAmount() {}
32
33    void testSetRecurring() {}
34
35 }

```

2. TASK: Write Test cases of remained generation login (Should trigger 2 days before due date)

```

Problems  @ Javadoc  Declaration  Console x  Progress  Terminal  Servers
<terminated> ReminderTest [JUnit] /home/jinwoo/.sdkman/candidates/java/17.0.10-tem/bin/java (Aug 1, :
Running: testReminderOnDueDate
Expected: false, Actual: false
Running: testReminderExactly2DaysBefore
Expected: true, Actual: true
Running: testReminder3DaysBefore
Expected: false, Actual: false
Running: testReminder1DayBefore
Expected: false, Actual: false

```

Code: *ReminderTest.java*

```
package paypilot;
import static org.junit.jupiter.api.Assertions.*;
import java.time.LocalDate;
import org.junit.jupiter.api.Test;
public class ReminderTest {
    @Test
    public void testReminderExactly2DaysBefore() {
        System.out.println("Running: testReminderExactly2DaysBefore");
        LocalDate dueDate = LocalDate.of(2025, 8, 10);
        LocalDate currentDate = LocalDate.of(2025, 8, 8);
        boolean result = Reminder.shouldSendReminder(dueDate, currentDate);
        System.out.println("Expected: true, Actual: " + result);
        assertTrue(result, "Reminder should be sent exactly 2 days before due date");
    }
    @Test
    public void testReminder1DayBefore() {
        System.out.println("Running: testReminder1DayBefore");
        LocalDate dueDate = LocalDate.of(2025, 8, 10);
        LocalDate currentDate = LocalDate.of(2025, 8, 9);
        boolean result = Reminder.shouldSendReminder(dueDate, currentDate);
        System.out.println("Expected: false, Actual: " + result);
        assertFalse(result, "Reminder should NOT be sent 1 day before");
    }
    @Test
    public void testReminder3DaysBefore() {
        System.out.println("Running: testReminder3DaysBefore");
        LocalDate dueDate = LocalDate.of(2025, 8, 10);
        LocalDate currentDate = LocalDate.of(2025, 8, 7);
        boolean result = Reminder.shouldSendReminder(dueDate, currentDate);
        System.out.println("Expected: false, Actual: " + result);
        assertFalse(result, "Reminder should NOT be sent 3 days before");
    }
    @Test
    public void testReminderOnDueDate() {
        System.out.println("Running: testReminderOnDueDate");
        LocalDate dueDate = LocalDate.of(2025, 8, 10);
        LocalDate currentDate = LocalDate.of(2025, 8, 10);
        boolean result = Reminder.shouldSendReminder(dueDate, currentDate);
        System.out.println("Expected: false, Actual: " + result);
        assertFalse(result, "Reminder should NOT be sent on the due date itself");
    }
}
```

3. TASK: Implement a function to validate if payment date is not before the due date, throw custom exception *InvalidPaymentDateException*.

```
@Test
public void testValidateDate() {
    assertThrows(InvalidPaymentDateException.class,
        () -> PaymentService.validatePaymentDate(bill, payment1)
    );
}
```

```

package com.paypilot.util;

import com.paypilot.exception.InvalidPaymentDateException;
import com.paypilot.model.Bill;
import com.paypilot.model.Payment;

public class PaymentDateValidator {

    public static void validatePaymentDate(Bill bill, Payment payment) throws InvalidPaymentDateException{
        if (payment.getPaymentDate().isBefore(bill.getDueDate())) {
            throw new InvalidPaymentDateException("Payment date cannot be before the due date.");
        } else {
            System.out.println("Payment date is valid." + payment.getPaymentDate());
        }
    }
}

```

4. TASK: Write parameterized tests to check valid bill categories.

The screenshot shows an IDE with two windows. The left window displays the test results for `BillCategoryParameterizedTest`. The test run finished after 0.211 seconds with 12/12 runs, 0 errors, and 2 failures. The test results are as follows:

Test Name	Duration	Status
testInvalidCategoriesShouldFail(String)	0.033 s	Failed
Invalid category test: Electricity	0.003 s	Failed
Invalid category test: Food	0.006 s	Failed
Invalid category test: Netflix	0.001 s	Failed
Invalid category test: Travel	0.001 s	Failed
Invalid category test: Gaming	0.000 s	Failed
Invalid category test: Shopping	0.001 s	Failed
testValidCategoriesShouldPass(String)	0.003 s	Passed
Valid category test: Electricity	0.003 s	Passed
Valid category test: Rent	0.000 s	Passed
Valid category test: Internet	0.001 s	Passed
Valid category test: Water	0.000 s	Passed
Valid category test: Gas	0.001 s	Passed
Valid category test: Grocery	0.003 s	Passed

The right window shows the source code for `BillCategoryParameterizedTest.java`:

```

4
5 import static org.junit.jupiter.api.Assertions.*;
6 import org.junit.jupiter.params.ParameterizedTest;
7 import org.junit.jupiter.params.provider.ValueSource;
8 import java.util.Arrays;
9 import java.util.List;
10
11 //Parameterized tests for validating allowed bill categories.
12 public class BillCategoryParameterizedTest {
13     // Defined the list of valid categories
14     private final List<String> VALID_CATEGORIES = Arrays.asList(
15         "Electricity", "Rent", "Internet", "Water", "Gas"
16     );
17
18     //Test that only known valid categories are accepted.
19     //Will fail for if any invalid category is passed (here-grocery)
20     @ParameterizedTest(name = "Valid category test: {0}")
21     @ValueSource(strings = {"Electricity", "Rent", "Internet", "Water", "Gas", "Grocery"})
22     void testValidCategoriesShouldPass(String category) {
23         assertTrue(VALID_CATEGORIES.contains(category),
24             () -> "Expected category '" + category + "' to be valid.");
25     }
26
27     //Tests the invalid categories are rejected.
28     //These tests will fail if the category accidentally exists in the valid list (here-electricity)
29     @ParameterizedTest(name = "Invalid category test: {0}")
30     @ValueSource(strings = {"Electricity", "Food", "Netflix", "Travel", "Gaming", "Shopping"})
31     void testInvalidCategoriesShouldFail(String category) {
32         assertFalse(VALID_CATEGORIES.contains(category),
33             () -> "Expected category '" + category + "' to be invalid.");
34     }
35 }

```

5. TASK: Write a method to check if a bill is recurring and autogenerate next month's bill, findAllBills()

```

// Method to find and return all bills
public List<Bill> findAllBills() {
    return new ArrayList<>(billList); //Returning a copy of Bill Lists to protect internal List
}

// Function to check if a bill is recurring
public static boolean isBillRecurring(Bill bill) {
    return bill != null && bill.isRecurring();
}

// Function to avoid generating a recurring bill if it already exists
public static boolean hasRecurringBillForNextMonth(Bill originalBill, List<Bill> allBills) {
    LocalDate nextDueDate = originalBill.getDueDate().plusMonths(1);

    return allBills.stream().anyMatch(existing ->
        existing.getUserId() == originalBill.getUserId() &&
        existing.getBillName().equalsIgnoreCase(originalBill.getBillName()) &&
        existing.getCategory().equalsIgnoreCase(originalBill.getCategory()) &&
        existing.getDueDate().equals(nextDueDate)
    );
}

```



```


// Function to check if a bill is recurring and autogenerating next months's bill
public static List<Bill> checkAndGenerateRecurringBills(List<Bill> bills)
{
    List<Bill> newBills = new ArrayList<>();
    Set<Integer> existingIds = new HashSet<>();
    LocalDate currentDate = LocalDate.now();
    //Adding existing IDs to the set
    for (Bill bill : bills)
    {
        existingIds.add(bill.getBillId());
    }
    for (Bill bill : bills)
    {
        if (bill.isRecurring() && bill.getDueDate().isBefore(currentDate))
        {
            // Generating a unique ID that doesn't clash
            int newId;
            do {
                Random random = new Random();

                newId = random.nextInt(Integer.MAX_VALUE); // Ensures a non-negative int
            } while (existingIds.contains(newId));
            existingIds.add(newId);
            // Generating Date for the next month's bill
            LocalDate nextDueDate = bill.getDueDate().plusMonths(1);
            //Checking if there is a bill already generated for next month
            if (!hasRecurringBillForNextMonth(bill, bills))
            {
                // Creating bill for the next month
                Bill nextMonthBill = new Bill(
                    bill.getUserId(),
                    newId,
                    bill.getBillName(),
                    bill.getCategory(),
                    nextDueDate,
                    bill.getAmount(),
                    true
                );
                newBills.add(nextMonthBill);
            }
        }
    }
    return newBills;
}

```

6. TASK: Write an assert for throwing custom exceptions when invalid dates are given.

Runs: 6/6  Errors: 0  Failures: 0

>  DateUtilTest [Runner: JUnit 5] (0.002 s)


```

1 package com.paypilot.test;
2 import org.junit.jupiter.api.Test;
3 public class DateUtilTest {
4
5     //Valid date input should return correct LocalDate
6     @Test
7     void testParseValidDate() {
8         String validDate = "28-07-2025";
9         LocalDate expected = LocalDate.of(2025, 7, 28);
10
11         LocalDate result = DateUtil.parse(validDate);
12         assertEquals(expected, result, "Parsed date should match expected LocalDate");
13     }
14
15     //Invalid format (e.g. wrong order "yyyy-MM-dd") should throw exception
16     @Test
17     void testParseInvalidFormatThrowsException() {
18         String invalidDate = "2025-07-28";
19
20         Exception exception = assertThrows(InvalidArgumentException.class, () -> {
21             DateUtil.parse(invalidDate);
22         });
23
24         assertTrue(exception.getMessage().contains("Invalid date format"), "Should contain 'Invalid date format'");
25     }
26
27     //Invalid date string like "invalid-date" should throw exception
28     @Test
29     void testParseGarbageDateThrowsException() {
30         String garbage = "invalid-date";
31
32         Exception exception = assertThrows(InvalidArgumentException.class, () -> {
33             DateUtil.parse(garbage);
34         });
35
36         assertTrue(exception.getMessage().contains("Invalid date format"), "Should contain 'Invalid date format'");
37     }
38
39     //Null string should throw exception with specific message
40     @Test
41     void testParseNullDateThrowsException() {
42         Exception exception = assertThrows(InvalidArgumentException.class, () -> {
43             DateUtil.parse(null);
44         });
45
46         assertEquals("Input date string cannot be null", exception.getMessage());
47     }
48
49     //Null LocalDate in format() should throw exception
50     @Test
51     void testFormatNullDateThrowsException() {
52         Exception exception = assertThrows(InvalidArgumentException.class, () -> {
53             DateUtil.format(null);
54         });
55
56         assertEquals("Input date cannot be null", exception.getMessage());
57     }
58
59     //Valid LocalDate should return correct formatted string
60     @Test
61     void testFormatValidDate() {
62         LocalDate date = LocalDate.of(2025, 7, 28);
63         String expected = "28-07-2025";
64
65         String result = DateUtil.format(date);
66         assertEquals(expected, result, "Formatted string should match expected pattern");
67     }
68 }

```

7. Create a method to group bills by category and return a `Map<String, List<Bill>>`. Create `getABillById()`.

```

// Group a group of bills by category
public static Map<String, List<Bill>> groupBillsByCategory(List<Bill> bills) {
    return bills.stream()
        .collect(Collectors.groupingBy(Bill::getCategory));
}

// Retrieves a bill based on bill id
public static Bill getABillById(List<Bill> bills, int billId) {
    for (Bill bill : bills) {
        if (bill.getBillId() == billId) {
            return bill;
        }
    }
    return null;
}

```

Report HACKATHON 2: 1st August, 2025

1. TASK

Create a derived column `late_fee` in a view, calculated as 10% of amount if `is_paid = false` and `due_date`.

Query:

```
CREATE OR REPLACE VIEW bills_with_late_fee AS
SELECT
  b.*,
  CASE
    WHEN b.is_paid = 0 AND b.due_date < SYSDATE
    THEN ROUND(b.amount * 0.10, 2)
    ELSE 0.00
  END AS late_fee
FROM bills b;
```

2. TASK

Create bills table with `bill_id`(String, primary key) , `bill_name` (String), `bill_category`(String), `due_date` (date), `amount`(float), `reminder_frequency` (String), `attachment` (String), `notes` (String), `is_recurring` (boolean), `is_paid` (boolean), `overdue_days` (int), `user_id`(String, Foreign key)

- `user_id` is a foreign key.
- Insert 5 bills: include 2 categories (electricity, rent) and future/past due dates.
- Add a unique constraint so that a user cannot have more than one bill with the same `due_date` & category.

Query:

```
CREATE TABLE Bills (
  bill_id VARCHAR2(50) PRIMARY KEY,
  bill_name VARCHAR2(100),
  bill_category VARCHAR2(50),
  due_date DATE,
  amount FLOAT,
  reminder_frequency VARCHAR2(50),
  attachment VARCHAR2(255),
  notes VARCHAR2(255),
  is_recurring NUMBER(1) CHECK (is_recurring IN (0, 1)),
  is_paid NUMBER(1) CHECK (is_paid IN (0, 1)),
  overdue_days NUMBER(3),
  user_id VARCHAR2(50),
  CONSTRAINT uq_user_due_cat UNIQUE (user_id, due_date, bill_category)
);
```

```
INSERT INTO Bills VALUES (  
    'B001', 'Ichigo Kurosaki', 'rent', TO_DATE('2025-07-28', 'YYYY-MM-DD'), 12000,  
    'monthly', NULL, 'Paid in advance', 0, 1, 0, 'U001'  
);
```

```
INSERT INTO Bills VALUES (  
    'B002', 'Tanjiro', 'rent', TO_DATE('2025-08-05', 'YYYY-MM-DD'), 12000,  
    'monthly', NULL, 'To be paid soon', 1, 0, 0, 'U001'  
);
```

```
INSERT INTO Bills VALUES (  
    'B003', 'Levi', 'electricity', TO_DATE('2025-07-30', 'YYYY-MM-DD'), 2500,  
    'monthly', NULL, 'Late payment expected', 1, 0, 2, 'U002'  
);
```

```
INSERT INTO Bills VALUES (  
    'B004', 'Artyom', 'electricity', TO_DATE('2025-08-02', 'YYYY-MM-DD'), 2700,  
    'monthly', NULL, 'Regular bill', 1, 0, 0, 'U003'  
);
```

```
INSERT INTO Bills VALUES (  
    'B005', 'Eren', 'rent', TO_DATE('2025-08-03', 'YYYY-MM-DD'), 12000,  
    'monthly', NULL, 'Test entry', 0, 0, 1, 'U003'  
);
```

3. TASK

Write a query to get all unpaid bills due within 5 days.

Query:

```
SELECT * FROM Bills  
WHERE is_paid = 0 AND due_date BETWEEN TRUNC(SYSDATE) AND TRUNC(SYSDATE) +  
5;
```

4. TASK

Write a query to join users and bills and display user name, bill category, and due date.

Query:

```
Select  
    u.name,  
    b.bill_category,  
    b.due_date  
from users u join Bills b  
on u.user_id=b.user_id
```


HACKATHON 2 OUTPUT

Table `BILLS` created.

1 row inserted.

1 row inserted.

1 row inserted.

1 row inserted.

1 row inserted.

	BILL_ID	BILL_NAME	BILL_CATEGORY	DUE_DATE	AMOUNT	REMINDER_FREQUENCY	ATTACHMENT	NOTES	IS_RECURRING	IS_PAID	OVERDUE_DAYS	USER_ID
1	0002	Tanjiro	rent	05/08/25	12000	monthly	(null)	To be paid soon	1	0	0	0001
2	0004	Artyom	electricity	02/08/25	2700	monthly	(null)	Regular bill	1	0	0	0003
3	0005	Eren	rent	03/08/25	12000	monthly	(null)	Test entry	0	0	1	0003

View `BILLS_WITH_LATE_FEE` created.

```
SQL> select
  2     u.name,
  3     b.bill_category,
  4     b.due_date
  5 from users u join Bills b
  6 on u.user_id=b.user_id;
```

NAME

BILL_CATEGORY

DUE_DATE

Sanjay Kumar
rent

28-JUL-25

Sanjay Kumar
rent

05-AUG-25

Priya Sharma
electricity

30-JUL-25

NAME

BILL_CATEGORY

DUE_DATE

Amit Singh
electricity

02-AUG-25

Amit Singh
rent

03-AUG-25