VARIOUS APPROACHES FOR PATTERN MATCHING CODE

Previous approach:

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        System.out.print("enter the number: ");
        int n = sc.nextInt();

        generatePattern(n);
    }

    public static void generatePattern(int n) {
        int[][] matrix = new int[n][n];
        int num = 1;

        for (int i = 0; i < n; i++) {
            int row;
            if (i % 2 == 0) {
                row = i / 2;
            } else {
                row = n - 1 - i / 2;
            }
            for (int j = 0; j < n; j++) {
```

```
                matrix[row][j] = num++;
            }
        }

        printMatrix(matrix, n);
    }

    public static void printMatrix(int[][] matrix, int n) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.print(matrix[i][j] + "\t");
            }
            System.out.println();
        }
    }
}
```

1. Direct Filling Using Incremental Index

Instead of assigning values row by row dynamically, directly fill the matrix while maintaining the pattern.

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number: ");
        int n = sc.nextInt();
        generatePattern(n);
    }

    public static void generatePattern(int n) {
        int[][] matrix = new int[n][n];
        int num = 1;

        for (int i = 0; i < n; i++) {
            int row = (i % 2 == 0) ? i / 2 : n - 1 - i / 2;
            for (int j = 0; j < n; j++) {
                matrix[row][j] = num++;
            }
        }

        printMatrix(matrix);
    }

    public static void printMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int val : row) {
                System.out.print(val + "\t");
            }
            System.out.println();
        }
    }
}
```

Optimization:

- **Uses ternary operator** for better readability.
- **Enhanced for-loops in printMatrix** (improves readability).
- **Removes extra n argument in printMatrix()**, as it's already known from matrix.length.

2. Using a Single-Dimensional Array

Instead of a 2D matrix, use a 1D array to store values and print them in the required format.

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number: ");
        int n = sc.nextInt();
        generatePattern(n);
    }

    public static void generatePattern(int n) {
        int[] arr = new int[n * n];
        int num = 1;

        for (int i = 0; i < n; i++) {
            int row = (i % 2 == 0) ? i / 2 : n - 1 - i / 2;
            for (int j = 0; j < n; j++) {
                arr[row * n + j] = num++;
            }
        }

        printArray(arr, n);
    }
```

```
    public static void printArray(int[] arr, int n) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + "\t");
            if ((i + 1) % n == 0) System.out.println();
        }
    }
}
```

Optimization:

- **Reduces memory overhead** by using a 1D array.
- **Improves cache efficiency**, as 1D arrays are more CPU-cache friendly.
- **Eliminates explicit row iteration in printMatrix**, as we calculate row breaks using i % n == 0.

3. Filling in a Zig-Zag Manner (Avoiding if checks)

Instead of determining row indices dynamically, we can predefine the pattern and follow it.

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number: ");
        int n = sc.nextInt();
        generatePattern(n);
    }
```

```
    public static void generatePattern(int n) {
        int[][] matrix = new int[n][n];
        int num = 1;

        int top = 0, bottom = n - 1;
        boolean toggle = true;

        for (int i = 0; i < n; i++) {
            int row = toggle ? top++ : bottom--;
            for (int j = 0; j < n; j++) {
                matrix[row][j] = num++;
            }
            toggle = !toggle;
        }

        printMatrix(matrix);
    }

    public static void printMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int val : row) {
                System.out.print(val + "\t");
            }
            System.out.println();
        }
    }
}
```

Optimization:

- **Avoids the modulus (%) operation** inside the loop.

- **Uses two pointers (top and bottom)** instead of recalculating row indices.
- **Boolean toggle switch** eliminates the need for ternary operators.

4. Using Bitwise Operations for Faster Even/Odd Check

We can replace i % 2 == 0 with (i & 1) == 0 for better performance.

```
public static void generatePattern(int n) {
    int[][] matrix = new int[n][n];
    int num = 1;

    for (int i = 0; i < n; i++) {
        int row = ((i & 1) == 0) ? i / 2 : n - 1 - i / 2;
        for (int j = 0; j < n; j++) {
            matrix[row][j] = num++;
        }
    }

    printMatrix(matrix);
}
```

EXCEPTION HANDLING

Exception handling in Java allows developers to manage runtime errors effectively by using mechanisms like try-catch block, finally block, throwing Exceptions, Custom Exception handling, etc.

An Exception is an unwanted or unexpected event that occurs during the execution of a program (i.e., at runtime) and disrupts the normal flow of the program's instructions. It occurs when something unexpected things happen, like accessing an invalid index, dividing by zero, or trying to open a file that does not exist.

Exception in Java is an error condition that occurs when something wrong happens during the program execution.

Types of Java Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.

Exceptions can be categorized in two ways:

1. **Built-in Exceptions**
   - Checked Exception
   - Unchecked Exception
2. **User-Defined Exceptions**

1. Built-in Exception

Build-in Exception are pre-defined exception classes provided by Java to handle common errors during program execution.

1.1 Checked Exceptions

Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler. Examples of Checked Exception are listed below:

1. **ClassNotFoundException:** Throws when the program tries to load a class at runtime but the class is not found because its not present in the correct location or it is missing from the project.

2. **InterruptedException:** Thrown when a thread is paused and another thread interrupts it.
3. **IOException:** Throws when input/output operation fails
4. **InstantiationException:** Thrown when the program tries to create an object of a class but fails because the class is abstract, an interface, or has no default constructor.
5. **SQLException:** Throws when there's an error with the database.
6. **FileNotFoundException:** Thrown when the program tries to open a file that doesn't exist

1.2 Unchecked Exceptions

The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Examples of Unchecked Exception are listed below:

1. **ArithmeticException:** It is thrown when there's an illegal math operation.
2. **ClassCastException:** It is thrown when you try to cast an object to a class it does not belongs to.
3. **NullPointerException:** It is thrown when you try to use a null object (e.g. accessing its methods or fields)
4. **ArrayIndexOutOfBoundsException:** It occurs when we try to access an array element with an invalid index.
5. **ArrayStoreException:** It **h**appens when you store an object of the wrong type in an array.
6. **IllegalThreadStateException:** It is thrown when a thread operation is not allowed in its current state

2. User-Defined Exception

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called "user-defined exception".

Try-Catch Block

A try-catch block in Java is a mechanism to handle exception. The try block contains code that might thrown an exception and the catch block is used to handles the exceptions if it occurs.

finally Block

The finally Block is used to execute important code regardless of weather an exception occurs or not.

throw Keyword

The throw keyword is used to manually throw an exception.

throw new CustomException("Message");

throws Keyword

The throws keyword is used in a method declaration to indicate that a method can throw exceptions.

```
public void myMethod() throws IOException {
    // Code that may throw an IOException
}
```

Creating Custom Exceptions

You can create your own exceptions by extending the Exception class or any of its subclasses. This allows you to define exception conditions that are specific to your application.

Example of Custom Exception:

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message); // Pass the message to the superclass constructor
    }
}
```

You can then use this custom exception as follows:

```
public void checkAge(int age) throws InvalidAgeException {
    if (age < 18) {
        throw new InvalidAgeException("Age must be at least 18.");
    }
}
```

Handling Multiple Exception:

We can handle multiple type of exceptions in Java by using multiple catch blocks, each catching a different type of exception.

```
try {
// Code that may throw an exception
} catch (ArithmeticException e) {
// Code to handle the exception
} catch(ArrayIndexOutOfBoundsException e){
//Code to handle the anothert exception
}catch(NumberFormatException e){
//Code to handle the anothert exception
}
```

Advantages of Exception Handling

- Provision to Complete Program Execution
- Easy Identification of Program Code and Error-Handling Code
- Propagation of Errors
- Meaningful Error Reporting
- Identifying Error Types

Program that includes types of exception: banking system

```
import java.io.*;
import java.util.*;


class InsufficientBalanceException extends Exception {
    public InsufficientBalanceException(String message) {
        super(message);
    }
```

```
}

class AccountNotFoundException extends Exception {
    public AccountNotFoundException(String message) {
        super(message);
    }
}

public class BankingSystem {

    static class Account {
        int accountNumber;
        String accountHolderName;
        double balance;

        public Account(int accountNumber, String accountHolderName, double balance) {
            this.accountNumber = accountNumber;
            this.accountHolderName = accountHolderName;
            this.balance = balance;
        }

        public void deposit(double amount) {
            balance += amount;
            System.out.println("Deposited: " + amount);
            System.out.println("New balance: " + balance);
```

```
    }

    public void withdraw(double amount) throws InsufficientBalanceException {
        if (balance < amount) {
            throw new InsufficientBalanceException("Insufficient balance for withdrawal.");
        }
        balance -= amount;
        System.out.println("Withdrawn: " + amount);
        System.out.println("New balance: " + balance);
    }

    public void displayAccountInfo() {
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Account Holder: " + accountHolderName);
        System.out.println("Current Balance: " + balance);
    }
}

public static Account getAccountDetails(int accountNumber) throws IOException,
AccountNotFoundException {
    Map<Integer, Account> accounts = new HashMap<>();
    accounts.put(101, new Account(101, "Alice", 5000.0));
    accounts.put(102, new Account(102, "Bob", 3000.0));
    accounts.put(103, new Account(103, "Charlie", 1000.0));
```

```
        Account account = accounts.get(accountNumber);
        if (account == null) {
            throw new AccountNotFoundException("Account with number " + accountNumber + " not found.");
        }

        return account;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter account number: ");
            int accountNumber = scanner.nextInt();
            Account account = getAccountDetails(accountNumber);
            account.displayAccountInfo();

            System.out.print("Enter amount to deposit: ");
            double depositAmount = scanner.nextDouble();
            account.deposit(depositAmount);

            System.out.print("Enter amount to withdraw: ");
            double withdrawalAmount = scanner.nextDouble();
            account.withdraw(withdrawalAmount);
```

```
        } catch (IOException e) {
            System.out.println("Error reading account data: " + e.getMessage());
        } catch (AccountNotFoundException e) {
            System.out.println("Error: " + e.getMessage());
        } catch (InsufficientBalanceException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            System.out.println("Transaction completed.");
            scanner.close();
        }
    }
}
```

Memory Allocation in Exception Handling

Exception handling in Java involves **heap memory allocation**, **stack memory usage**, and **object creation**.

(a) Heap Memory (Object Allocation)

- When an exception object is created using new Exception("message"), it is stored in **heap memory**.
- This object stays in memory until the Garbage Collector removes it.

Example:

Exception e = new Exception("Something went wrong!"); // Allocated in Heap

(b) Stack Memory (Method Calls & Exception Propagation)

- Java maintains a **call stack** for each thread.
- When an exception is thrown, it **unwinds the call stack** to find a matching catch block.
- Each method call (including exception handlers) gets **stack space**.

Example of how exceptions affect stack memory:

```
public class StackMemoryExample {
    public static void main(String[] args) {
        try {
            methodA();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void methodA() {
        methodB();
    }

    static void methodB() {
        methodC();
    }

    static void methodC() {
        throw new RuntimeException("Exception at C");
    }
}
```

Stack Memory Flow:

main() -> methodA() -> methodB() -> methodC()

- methodC() throws an exception.
- Stack unwinds back to main(), where the exception is caught.
- Stack frames (methodC, methodB, methodA) are removed as the exception propagates.

Summary Table: Exception Handling and Memory Allocation

Component	Memory Location	Impact
Exception Object	Heap Memory	Stays until garbage collected
Stack Trace	Heap/Stack Memory	Takes extra memory if printed
Call Stack Frames	Stack Memory	Removed as exception propagates
Catch Blocks	Stack Memory	Stored in stack until execution

Ticket purchase

Flow:

1. **View Events**
    - Show available events (movies, concerts, trains, flights, etc.).
    - Display details: event name, date, time, price, and available seats.
2. **Select an Event**
    - User selects an event by entering an ID or name.
    - Show seating options (if applicable).
3. **Enter Ticket Details**
    - Number of tickets.
    - Category (Standard, VIP, etc.).
4. **Payment Process**
    - Select payment method (Credit/Debit, UPI, Cash).
    - Enter necessary details.
    - Process payment.
5. **Generate Ticket**
    - Display ticket details (Event, Seat No, Price, Transaction ID).
    - Option to print or email the ticket.
6. **Exit or Continue**
    - Return to the main menu or exit the app.

Implementation Plan

Classes Needed

1. Event – Stores event details.
2. Ticket – Stores ticket details.
3. Payment – Handles payments.
4. TicketSystem – Main class that controls the program.