**DAY14 – July 04th**

**TASK-1 – Create a Node and push elements into linkedlist, traverse and display the elements.**

package Day14;  
  
class Node6{  
 int data;  
 Node6 next;  
  
 Node6(int data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
  
public class TASK1 {  
 Node6 head;  
  
 public void push(int data) {  
 Node6 newNode = new Node6(data);  
  
 if (head == null) {  
 head = newNode;  
 return;  
 }  
  
 Node6 curr = head;  
 while (curr.next != null) {  
 curr = curr.next;  
 }  
  
 curr.next = newNode;  
 }  
  
 public void display() {  
 Node6 current = head;  
 System.*out*.print("Linked List Elements: ");  
 while (current != null) {  
 System.*out*.print(current.data + " ");  
 current = current.next;  
 }  
 System.*out*.println();  
 }  
  
 public static void main(String[] args) {  
 TASK1 list = new TASK1();  
  
 // Push 4 elements  
 list.push(10);  
 list.push(20);  
 list.push(30);  
 list.push(40);  
  
 // Display the list  
 list.display();  
 }  
}

**Output**

Linked List Elements: 10 20 30 40

**TASK2 – What do you mean by Traversing?**

**Traversing a linked list** means **visiting each node** in the list **one by one**, starting from the **head** (first node), and moving through the next pointers until you reach the end (null).

* **Print** all elements
* **Search** for a value
* **Count** nodes
* **Update** data in nodes
* **Perform operations** like sum, max, etc.

**TASK3 – Create Circular Linked List**

package Day14;  
  
class Node7 {  
 int data;  
 Node7 next;  
  
 Node7(int data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
  
class Task3\_CircularLinkedLIst {  
 Node7 head = null;  
 Node7 tail = null;  
  
 // Method to add node at the end (tail)  
 public void push(int data) {  
 Node7 newNode = new Node7(data);  
  
 if (head == null) {  
 // First node: points to itself  
 head = newNode;  
 tail = newNode;  
 newNode.next = head;  
 } else {  
 // Add after tail and make new node the new tail  
 tail.next = newNode;  
 tail = newNode;  
 tail.next = head; // Make it circular  
 }  
 }  
  
 // Method to display circular linked list  
 public void display() {  
 if (head == null) {  
 System.*out*.println("List is empty.");  
 return;  
 }  
  
 Node7 current = head;  
 System.*out*.print("Circular Linked List Elements: ");  
 do {  
 System.*out*.print(current.data + " ");  
 current = current.next;  
 } while (current != head); // Stop after one full loop  
 System.*out*.println();  
 }  
  
 public static void main(String[] args) {  
 Task3\_CircularLinkedLIst cll = new Task3\_CircularLinkedLIst();  
  
 // Insert 4 elements  
 cll.push(10);  
 cll.push(20);  
 cll.push(30);  
 cll.push(40);  
  
 // Display the circular list  
 cll.display();  
 }  
}

**Output**

Circular Linked List Elements: 10 20 30 40

**TASK4 - Delete a node in Circular Linked List**

package Day14;  
  
class Node8 {  
 int data;  
 Node8 next;  
  
 Node8(int data) {  
 this.data = data;  
 }  
}  
  
public class TASK4\_DeleteNode {  
 Node8 head = null;  
 Node8 tail = null;  
  
 // Method to add node at end  
 public void push(int data) {  
 Node8 newNode1 = new Node8(data);  
 if (head == null) {  
 head = newNode1;  
 tail = newNode1;  
 newNode1.next = head;  
 } else {  
 tail.next = newNode1;  
 tail = newNode1;  
 tail.next = head;  
 }  
 }  
  
 // Alternative method to delete a node  
 public void delete2(int key) {  
 if (head == null) {  
 System.*out*.println("List is empty.");  
 return;  
 }  
  
 // Case 1: Only one node in the list  
 if (head == tail && head.data == key) {  
 head = null;  
 tail = null;  
 System.*out*.println("Deleted node with value: " + key);  
 return;  
 }  
  
 // Case 2: Deleting the head  
 if (head.data == key) {  
 head = head.next;  
 tail.next = head;  
 System.*out*.println("Deleted node with value: " + key);  
 return;  
 }  
  
 // Case 3: Deleting any node other than head  
 Node8 current = head;  
 while (current.next != head) {  
 if (current.next.data == key) {  
 if (current.next == tail) {  
 tail = current;  
 }  
 current.next = current.next.next;  
 System.*out*.println("Deleted node with value: " + key);  
 return;  
 }  
 current = current.next;  
 }  
  
 System.*out*.println("Node with value " + key + " not found.");  
 }  
  
 public void display() {  
 if (head == null) {  
 System.*out*.println("List is empty.");  
 return;  
 }  
  
 Node8 current = head;  
 System.*out*.print("Circular Linked List Elements: ");  
 do {  
 System.*out*.print(current.data + " ");  
 current = current.next;  
 } while (current != head);  
 System.*out*.println();  
 }  
  
 public static void main(String[] args) {  
 TASK4\_DeleteNode cll = new TASK4\_DeleteNode();  
  
 cll.push(10);  
 cll.push(20);  
 cll.push(30);  
 cll.push(40);  
  
 cll.display();  
  
 cll.delete2(10); // Delete head  
 cll.display();  
  
 cll.delete2(40); // Delete tail  
 cll.display();  
  
 cll.delete2(25); // Not found  
 cll.display();  
  
 cll.delete2(20);  
 cll.delete2(30); // All deleted  
 cll.display();  
 }  
}

**Output**

Circular Linked List Elements: 10 20 30 40

Deleted node with value: 10

Circular Linked List Elements: 20 30 40

Deleted node with value: 40

Circular Linked List Elements: 20 30

Node with value 25 not found.

Circular Linked List Elements: 20 30

Deleted node with value: 20

Deleted node with value: 30

List is empty.

**TASK5 – Create Stack using PUSH and POP Operations?**

package Day14;  
import java.util.Stack;  
public class TASK5\_PredfinedStack {  
 public static void main(String[] args) {  
 Stack<Integer> stack = new Stack<>();  
  
 stack.push(10);  
 stack.push(20);  
 stack.push(30);  
  
 System.*out*.println("Stack: " + stack); // [10, 20, 30]  
  
 // Peek (top of stack)  
 System.*out*.println("Top element: " + stack.peek()); // 30  
  
 // Pop  
 System.*out*.println("Popped: " + stack.pop()); // 30  
 System.*out*.println("Stack after pop: " + stack); // [10, 20]  
 System.*out*.println("Popped: " + stack.pop()); // 20  
 System.*out*.println("Stack after pop: " + stack); // [10, 20]  
 }  
}

**Output**

Stack: [10, 20, 30]

Top element: 30

Popped: 30

Stack after pop: [10, 20]

Popped: 20

Stack after pop: [10]

**TASK6 - Find an element in the stack and display the position**

package Day14;  
  
import java.util.Stack;  
  
public class TASK6\_StackSearch {  
  
 public static void main(String[] args) {  
 // Creating a stack of integers  
 Stack<Integer> stack = new Stack<>();  
  
 // Pushing elements into the stack  
 stack.push(10);  
 stack.push(20);  
 stack.push(30);  
 stack.push(40);  
  
 // Displaying the stack  
 System.*out*.println("Current Stack: " + stack); // [10, 20, 30, 40]  
  
 // Searching for an element  
 int searchElement = 30;  
 int position = stack.search(searchElement);  
 System.*out*.println("Element found at position : " + position);  
  
 if (position != -1) {  
 System.*out*.println("Element " + searchElement + " found at position : " + position);  
 }  
 else  
 {  
 System.*out*.println("Element " + searchElement + " not found in the stack.");  
 }  
 }  
}

**Output**

Current Stack: [10, 20, 30, 40]

Element found at position: 2

Element 30 found at position: 2

package Day14;  
  
import java.util.Stack;  
  
public class TASK6\_StackSearch {  
  
 public static void main(String[] args) {  
 Stack<Integer> stack = new Stack<>();  
  
 stack.push(10);  
 stack.push(20);  
 stack.push(30);  
 stack.push(40);  
  
 System.*out*.println("Current Stack: " + stack); // [10, 20, 30, 40]  
  
 int searchElement = 30;  
 int position = stack.search(searchElement);  
 System.*out*.println("Element found at position : " + position);  
 System.*out*.println(stack.search(10));  
 }  
}

**Output**

Current Stack: [10, 20, 30, 40]

Element found at position: 2

4

**TASK7 – Peek the element and print it.**

package Day14;  
  
import java.util.Stack;  
public class TASK7\_StackPeek {  
 public static void main(String[] args) {  
 Stack<Integer> stack = new Stack<>();  
  
 stack.push(100);  
 stack.push(200);  
 stack.push(300);  
  
 System.*out*.println("Stack: " + stack);// [100, 200, 300]  
 System.*out*.println("Peek element : " + stack.peek());  
 }  
}

**Output**

Stack: [100, 200, 300]

Peek element: 300

**TASK8 – Stack Operations**

package Day14;  
import java.util.Stack;  
public class TASK8\_StackOperations {  
 public static void main(String[] args) {  
 Stack<Integer> stack = new Stack<>();  
  
 // Check if the stack is empty initially  
 System.*out*.println("Is stack empty? " + stack.isEmpty()); // true  
 stack.push(10);  
 stack.push(20);  
 stack.push(30);  
 System.*out*.println("Stack after pushing: " + stack); // [10, 20, 30]  
  
 System.*out*.println("Is stack empty now? " + stack.isEmpty()); // false  
  
 int poppedElement = stack.pop();  
 System.*out*.println("Popped element: " + poppedElement); // 30  
 int poppedElement1 = stack.pop();  
 System.*out*.println("Popped element: " + poppedElement1);  
 int poppedElement2 = stack.pop();  
 System.*out*.println("Popped element: " + poppedElement2);  
 System.*out*.println("Stack after pop: " + stack); // [10, 20]  
  
 if(stack.isEmpty())  
 {  
 System.*out*.println("Stack is Empty");  
 }  
 }  
}

**Output**

Is stack empty? true

Stack after pushing: [10, 20, 30]

Is stack empty now? false

Popped element: 30

Popped element: 20

Popped element: 10

Stack after pop: []

Stack is Empty

**TASK9 – Methods in Stack Class**

| Method | Description |
| --- | --- |

|  |  |
| --- | --- |
| push(E item) | Adds (pushes) an item onto the top of the stack. |

|  |  |
| --- | --- |
| pop() | Removes and returns the top element of the stack. |

|  |  |
| --- | --- |
| peek() | Returns the top element without removing it. |

|  |  |
| --- | --- |
| search(Object o) | Returns the 1-based position from the top of the stack; -1 if not found. |

|  |  |
| --- | --- |
| empty() | Checks if the stack is empty (deprecated; prefer isEmpty()). |

|  |  |
| --- | --- |
| isEmpty() | Checks whether the stack is empty (inherited from Vector). |

|  |  |
| --- | --- |
| size() | Returns the number of elements in the stack. |

|  |  |
| --- | --- |
| clear() | Removes all elements from the stack. |

|  |  |
| --- | --- |
| contains(Object o) | Returns true if the stack contains the specified element. |

|  |
| --- |
| get(int index) |

Returns the element at the specified index (0-based).

**TASK10 – QueueOperations**

package Day14;  
  
public class TASK10\_QueueOperations {  
 int front = -1;  
 int rear = -1;  
 int size;  
 int[] queue;  
  
 public TASK10\_QueueOperations(int size) {  
 this.size = size;  
 queue = new int[size];  
 }  
  
 // Check if the queue is empty  
 public boolean isEmpty() {  
 return front == -1 || front > rear;  
 }  
  
 // Check if the queue is full  
 public boolean isFull() {  
 return rear == size - 1;  
 }  
  
 // Enqueue (add element)  
 public void enqueue(int data) {  
 if (isFull()) {  
 System.*out*.println("Queue is full. Cannot enqueue " + data);  
 return;  
 }  
  
 if (isEmpty()) {  
 front = 0;  
 }  
  
 rear++;  
 queue[rear] = data;  
 System.*out*.println("Enqueued: " + data);  
 }  
  
 // Dequeue (remove element)  
 public void dequeue() {  
 if (isEmpty()) {  
 System.*out*.println("Queue is empty. Cannot dequeue.");  
 return;  
 }  
  
 System.*out*.println("Dequeued: " + queue[front]);  
 front++;  
 }  
  
 // Peek (front element)  
 public void peek() {  
 if (isEmpty()) {  
 System.*out*.println("Queue is empty. No front element.");  
 } else {  
 System.*out*.println("Front element: " + queue[front]);  
 }  
 }  
  
 // Display all elements  
 public void display() {  
 if (isEmpty()) {  
 System.*out*.println("Queue is empty.");  
 return;  
 }  
  
 System.*out*.print("Queue elements: ");  
 for (int i = front; i <= rear; i++) {  
 System.*out*.print(queue[i] + " ");  
 }  
 System.*out*.println();  
 }  
  
 // Main method to test  
 public static void main(String[] args) {  
 TASK10\_QueueOperations q = new TASK10\_QueueOperations(5);  
  
 q.enqueue(10);  
 q.enqueue(20);  
 q.enqueue(30);  
 q.display();  
  
 q.peek(); // Show front  
 q.dequeue(); // Remove front  
 q.display(); // After dequeue  
  
 q.enqueue(40);  
 q.enqueue(50);  
 q.enqueue(60); // Should fill the queue  
 q.enqueue(70); // Should say "Queue is full"  
  
 q.display();  
 System.*out*.println("Is queue full? " + q.isFull());  
 System.*out*.println("Is queue empty? " + q.isEmpty());  
 }  
}

**Output**

Enqueued: 10

Enqueued: 20

Enqueued: 30

Queue elements: 10 20 30

Front element: 10

Dequeued: 10

Queue elements: 20 30

Enqueued: 40

Enqueued: 50

Queue is full. Cannot enqueue 60

Queue is full. Cannot enqueue 70

Queue elements: 20 30 40 50

Is queue full? true

Is queue empty? False

package Day14;  
  
class Node9{  
 int data;  
 Node9 next;  
  
 Node9(int data) {  
 this.data = data;  
 this.next = null;  
 }  
}  
  
public class TASK10\_1 {  
 Node9 front = null;  
 Node9 rear = null;  
 int size = 0;  
 int maxSize = 5; // Optional: set a maximum queue size  
  
 // Check if queue is empty  
 public boolean isEmpty() {  
 return front == null;  
 }  
  
 // Check if queue is full  
 public boolean isFull() {  
 return size == maxSize;  
 }  
  
 // Enqueue (add to rear)  
 public void enqueue(int data) {  
 if (isFull()) {  
 System.*out*.println("Queue is full. Cannot enqueue " + data);  
 return;  
 }  
  
 Node9 newNode = new Node9(data);  
  
 if (isEmpty()) {  
 front = rear = newNode;  
 } else {  
 rear.next = newNode;  
 rear = newNode;  
 }  
  
 size++;  
 System.*out*.println("Enqueued: " + data);  
 }  
  
 // Dequeue (remove from front)  
 public void dequeue() {  
 if (isEmpty()) {  
 System.*out*.println("Queue is empty. Cannot dequeue.");  
 return;  
 }  
  
 System.*out*.println("Dequeued: " + front.data);  
 front = front.next;  
 size--;  
  
 if (front == null) {  
 rear = null;  
 }  
 }  
  
 // Peek (see front element)  
 public void peek() {  
 if (isEmpty()) {  
 System.*out*.println("Queue is empty. No front element.");  
 } else {  
 System.*out*.println("Front element: " + front.data);  
 }  
 }  
  
 // Display all elements  
 public void display() {  
 if (isEmpty()) {  
 System.*out*.println("Queue is empty.");  
 return;  
 }  
  
 System.*out*.print("Queue elements: ");  
 Node9 current = front;  
 while (current != null) {  
 System.*out*.print(current.data + " ");  
 current = current.next;  
 }  
 System.*out*.println();  
 }  
  
 // Main method  
 public static void main(String[] args) {  
 TASK10\_1 queue = new TASK10\_1();  
  
 queue.enqueue(10);  
 queue.enqueue(20);  
 queue.enqueue(30);  
 queue.enqueue(40);  
 queue.enqueue(50);  
 queue.enqueue(60); // Should show full message  
  
 queue.display();  
  
 queue.peek(); // Front element  
  
 queue.dequeue();  
 queue.display();  
  
 System.*out*.println("Is queue empty? " + queue.isEmpty());  
 System.*out*.println("Is queue full? " + queue.isFull());  
 }  
}

**Output**

Enqueued: 10

Enqueued: 20

Enqueued: 30

Enqueued: 40

Enqueued: 50

Queue is full. Cannot enqueue 60

Queue elements: 10 20 30 40 50

Front element: 10

Dequeued: 10

Queue elements: 20 30 40 50

Is queue empty? false

Is queue full? False

**HOME TASK - converting stack and deque into a lists and printing their elements in java using streams.**

package Day14;  
  
import java.util.\*;  
import java.util.stream.Collectors;  
  
class HomeTask\_StackDequeStream {  
 public static void main (String[] args) {  
  
 Stack<Integer> stack = new Stack<>();  
 Deque<Integer> deque = new ArrayDeque<>();  
  
 stack.push(1);  
 deque.push(1);  
 stack.push(2);  
 deque.push(2);  
  
 List<Integer> list1 = stack.stream().collect(Collectors.*toList*());  
 System.*out*.println("Using Stack: ");  
 for(int i = 0; i < list1.size(); i++){  
 System.*out*.print(list1.get(i) + " " );  
 }  
 System.*out*.println();  
  
 List<Integer> list2 = deque.stream().collect(Collectors.*toList*());  
 System.*out*.println("Using Deque: ");  
 for(int i = 0; i < list2.size(); i++){  
 System.*out*.print(list2.get(i) + " " );  
 }  
 System.*out*.println();  
  
 }  
}

**Output**

Using Stack:

1 2

Using Deque:

2 1

**HOMETASK– Factorial**

package Day14;  
  
import java.util.Scanner;  
  
public class HomeTask\_FactorialRecursive {  
 // Recursive method to calculate factorial  
 static long factorial(int n) {  
 if (n == 0 || n == 1)  
 return 1;  
 return n \* *factorial*(n - 1);  
 }  
  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.print("Enter a number: ");  
 int n = sc.nextInt();  
  
 long result = *factorial*(n);  
 System.*out*.println("Factorial of " + n + " is: " + result);  
 }  
}

**Output**

Enter a number: 6

Factorial of 6 is: 720

**HOMETASK – Fibanocci Using Recursive**

package Day14;  
  
import java.util.Scanner;  
  
public class HomeTask\_FibanocciRecursive {  
 // Recursive function to return the nth Fibonacci number  
 static int fib(int n) {  
 if (n == 0)  
 return 0;  
 else if (n == 1)  
 return 1;  
 else  
 return *fib*(n - 1) + *fib*(n - 2);  
 }  
  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.print("Enter the number of terms: ");  
 int n = sc.nextInt();  
  
 System.*out*.println("Fibonacci Series:");  
 for (int i = 0; i < n; i++) {  
 System.*out*.print(*fib*(i) + " ");  
 }  
 }  
}

**Output**

Enter the number of terms: 8

Fibonacci Series:

0 1 1 2 3 5 8 13

**HOMETASK – Difference between Iteration and Recursion**

**Recursion**

* A function calls **itself** to solve a smaller version of the problem.
* It uses a **base case** to stop the recursion.
* Each recursive call is stored in the **call stack**, so it uses **more memory**.
* Can lead to **stack overflow** if too many recursive calls are made.
* Often easier to write for **tree**, **graph**, or **divide-and-conquer** problems.
* Code is usually **shorter and cleaner**, but may be **slower**.
* Good for problems like **factorial**, **Fibonacci**, **tower of Hanoi**, **tree traversal**, etc.

**Iteration**

* Uses **loops** (for, while, do-while) to repeat a block of code.
* Terminates when a **loop condition** becomes false.
* Uses **less memory** since no function calls are stored.
* Generally **faster and more efficient** than recursion.
* Better for simple, repeated tasks like **printing**, **summing**, or **searching**.
* Code can be **longer** but is usually **more efficient**.
* Commonly used when the number of repetitions is **known or predictable**.

**HOMETASK – Reverse a String using Recursion**

package Day14;  
public class HomeTask\_ReverseString {  
  
 // Recursive function to reverse a string  
 public static String reverse(String str) {  
 // Base case: if the string is empty or has 1 character  
 if (str.isEmpty()) {  
 return str;  
 }  
 // Recursive case: reverse the rest of the string and add the first character at the end  
 return *reverse*(str.substring(1)) + str.charAt(0);  
 }  
  
 public static void main(String[] args) {  
 String original = "hello";  
 String reversed = *reverse*(original);  
  
 System.*out*.println("Original string: " + original);  
 System.*out*.println("Reversed string: " + reversed);  
 }  
}

**Output**

Original string: hello

Reversed string: olleh

**HOMETASK - Write a recursive function to search for an element in an array?**

package Day14;  
  
public class HomeTask\_RecursiveSearch {  
  
 // Recursive function to search for an element in an array  
 public static int search(int[] arr, int target, int index) {  
 // Base case: if index exceeds array length  
 if (index >= arr.length) {  
 return -1; // element not found  
 }  
  
 // If current element matches the target  
 if (arr[index] == target) {  
 return index;  
 }  
  
 // Recursive case: search in the remaining part of the array  
 return *search*(arr, target, index + 1);  
 }  
  
 public static void main(String[] args) {  
 int[] arr = {5, 8, 12, 7, 20};  
 int target = 7;  
  
 int result = *search*(arr, target, 0); // Start from index 0  
  
 if (result != -1) {  
 System.*out*.println("Element " + target + " found at index " + result);  
 } else {  
 System.*out*.println("Element " + target + " not found.");  
 }  
 }  
}

**Output**

Element 7 found at index 3

**HOMETASK - Write a recursive function to count the digits of a positive integer.**

package Day14;  
  
public class HomeTask\_CountDigits {  
  
 // Recursive function to count digits  
 public static int countDigits(int number) {  
 // Base case: if number is 0, it has 0 digits (handled separately for 0 case)  
 if (number == 0) {  
 return 0;  
 }  
  
 // Recursive case: divide number by 10 and add 1 to the count  
 return 1 + *countDigits*(number / 10);  
 }  
  
 public static void main(String[] args) {  
 int num = 12345;  
  
 // Special case for 0  
 if (num == 0) {  
 System.*out*.println("Number of digits: 1");  
 } else {  
 int digitCount = *countDigits*(num);  
 System.*out*.println("Number of digits: " + digitCount);  
 }  
 }  
}

**Output**

Number of digits: 5

**HOMETASK - Write a recursive function to calculate the Sum Of Digits**

package Day14;  
  
public class HomeTask\_SumOfDigits {  
  
 // Recursive function to calculate sum of digits  
 public static int sumDigits(int number) {  
 // Base case: when number becomes 0  
 if (number == 0) {  
 return 0;  
 }  
  
 // Recursive case: add last digit + sum of remaining digits  
 return (number % 10) + *sumDigits*(number / 10);  
 }  
  
 public static void main(String[] args) {  
 int num = 1234;  
  
 int sum = *sumDigits*(num);  
 System.*out*.println("Sum of digits of " + num + " = " + sum);  
 }  
}

**Output**

Sum of digits of 1234 = 10

**HOMETASK - Write a recursive function to reverse a null-terminated string.**

package Day14;  
  
public class HomeTask\_NullString {  
  
 // Recursive function to reverse a string  
 public static String reverse(String str) {  
 // Base case: if the string is empty or has one character  
 if (str == null || str.length() <= 1) {  
 return str;  
 }  
  
 // Recursive case:  
 // last character + reverse of the substring excluding the last character  
 return str.charAt(str.length() - 1) + *reverse*(str.substring(0, str.length() - 1));  
 }  
  
 public static void main(String[] args) {  
 String input = "hello";  
 String reversed = *reverse*(input);  
 System.*out*.println("Reversed string: " + reversed);  
 }  
}

**Output**

Reversed string: olleh

**HOMETASK – Write a recursive function to convert a decimal number to binary**

package Day14;  
  
public class HomeTask\_DecimalToBinary {  
  
 // Recursive function to convert decimal to binary  
 public static void convertToBinary(int n) {  
 if (n == 0) {  
 return;  
 }  
  
 *convertToBinary*(n / 2); // recursive call with quotient  
 System.*out*.print(n % 2); // print remainder after recursion  
 }  
  
 public static void main(String[] args) {  
 int decimal = 13;  
  
 if (decimal == 0) {  
 System.*out*.print(0);  
 } else {  
 *convertToBinary*(decimal);  
 }  
 }  
}

**Output**

1101