****

**A REPORT**

*for*

**Mini Project (22CSE48)**

*on*

**Tic Tac Toe Application**

*Submitted by*

**C R Sandhya USN: 1NH22CS053, 4-A**

**Aajnya Narayan Prabhu USN: 1NH22CS004, 4-A**

*In partial fulfillment for the award of the degree of*

*in*

**COMPUTER SCIENCE AND ENGINEERING**

**Academic Year: 2023-24**

****

**CERTIFICATE**

This is to certify that the mini project work titled

**Tic Tac Toe Application**

submitted in partial fulfillment of the degree of Bachelor of Engineering in Computer Science and Engineering by

**C R Sandhya USN: 1NH22CS053, 4-A**

**Aajnya Narayan Prabhu USN: 1NH22CS004, 4-A**

*DURING*

*EVEN SEMESTER 2023-2024*

*for*

*Course: Mini Project-22CSE48*

Signature of Reviewer Signature of HOD

**SEMESTER END EXAMINATION**

*Name of the Examiner(s) Signature with date*

|  |  |
| --- | --- |
| 1. |  |
| 2. |  |

**ABSTRACT**

This project presents the development of a Tic-Tac-Toe application using Java, aimed at demonstrating core programming principles and object-oriented design. The application features a user-friendly graphical interface created with Java Swing, offering an interactive and engaging user experience. The game logic, encapsulated within modular classes, ensures maintainability and scalability. Key components include a dynamic game board, robust player management, and efficient win condition algorithms. This project underscores the importance of clean code practices, error handling, and comprehensive testing. Ultimately, it provides a practical and educational example of game development in Java, suitable for both novice and experienced programmers.

**ACKNOWLEDGEMENT**

The satisfaction and euphoria that accompany the successful completion of any task would be impossible without the mention of the people who made it possible, whose constant guidance and encouragement crowned our efforts with success.

We are delighted to express our gratitude to **Dr. Mohan Manghnani**, Chairman, New Horizon Educational Institutions, for furnishing the essential infrastructure and fostering a positive environment.

We would like to take this chance to express our deep gratitude to **Dr. Manjunatha, Principal**, New Horizon College of Engineering, for consistently offering support and encouragement.

We wish to convey our gratitude to **Dr. Anandhi R J**, Professor and Dean-Academics at NHCE, for providing indispensable guidance and unwavering support.

We want to express our heartfelt gratitude to **Dr. B.** **Rajalakshmi**, Professor and Head of the Department, Computer Science and Engineering, for the steadfast support that has remarkably shaped our academic journey.

We would like to extend our thanks to **Ms. Asha Rani Borah**, Senior Associate Professor, Department of Computer Science and Engineering, who served as the reviewer for our mini project.

**C R Sandhya**

**USN:1NH22CS053**

**Aajnya Narayan Prabhu**

**USN:1NH22CS004**

**CONTENTS**

**ABSTRACT I**

**ACKNOWLEDGEMENT II**

**LIST OF FIGURES VI**

1. **INTRODUCTION** 
   1. OVERVIEW **1**
   2. PROBLEM DEFINITION **1**
   3. OBJECTIVES **2**
   4. METHODOLOGY TO BE FOLLOWED **2**
   5. EXPECTED OUTCOMES **3**
   6. HARDWARE AND SOFTWARE REQUIREMENTS **4**
2. **FUNDAMENTALS OF JAVA**
   1. INTRODUCTION TO JAVA **5**
   2. ADVANTAGES OF JAVA **6**
   3. DATA TYPES **7**
   4. CONTROL FLOW **8**
   5. METHODS **9**
   6. OBJECT ORIENTED CONCEPTS **9**
   7. EXCEPTION HANDLING **10**
   8. FILE HANDLING **12**
   9. PACKAGES AND IMPORT **13**
   10. INTERFACES **13**
   11. CONCURRENCY **17**

1. **JAVA COLLECTIONS GUIDE**
   1. OVERVIEW OF JAVA COLLECTIONS **18**
   2. IMPORTANTS IN JAVA DEVELOPMENT **19**
   3. BENEFITS AND USECASES **20**
   4. CORE COLLECTION INTERFACES **21**
   5. COMMON COLLECTION IMPLEMENTATIONS **22**
   6. ITERATORS AND COLLECTIONS API **23**
   7. CUSTOM COLLECTIONS AND GENERICS **24**
2. **DESIGN AND ARCHITECTURE** 
   1. DESIGN GOALS **26**
   2. DATABASE STRUCTURE **27**
   3. HIGH LEVEL ARCHITECTURE **28**
   4. CLASS DIAGRAM **28**
3. **IMPLEMENTATION**
   1. CREATING THE MAIN WINDOW  **30**
   2. DISPLAYING FRAMES OVER THE MAIN WINDOW **30**
   3. PROCESSING QUERIES **30**
   4. CODING THE CORE FUNCTIONALITY **31**
   5. HANDLING USER INPUTS **31**
   6. ERROR HANDLING AND VALIDATION **32**
4. **TESTING** 
   1. UNIT TESTING **33**
   2. INTEGRATION TESTING **33**
   3. SYSTEM TESTING **34**
5. **RESULTS** 
   1. WELCOME PAGE **35**
   2. Symbol and mode selection page **35**
   3. Error if same symbol is selected by both the players **36**

7.4 Message after any player wins **36**

7.5. Message after computer wins (vs computer mode). **37**

7.6. MESSAGE AFTER GAME ENDS IN A TIE **37**

1. **CONCLUSION 38**

**REFERENCES 40**

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **Figure No** | **Figure Description** | **Page No** |
| 2.1 | Java | 5 |
| 2.10 | Interfaces | 13 |

4.4 Java architecture 29

7.1 Welcome page 35

7.2 Symbol and mode selection page 35

7.3 Error message 36

7.4 Win message 36

7.5 Win message (vs computer) 37

7.6 Tie message 37

**Chapter 1**

**INTRODUCTION**

Tic Tac Toe, also known as Noughts and Crosses or X’s and O’s, is a classic two-player game. This simple yet strategic game is a great way to introduce basic programming concepts, user interface design, and game logic. In this application, two players take turns marking spaces in a 3x3 grid The aim is to be the primary to vicinity 3 in their marks in a horizontal, vertical, or diagonal row. Despite its straightforward rules, Tic Tac Toe offers opportunities to explore important programming topics such as user interface design, game state management, and win/draw condition detection.

**1.1 Overview**

The Tic Tac Toe game is a well-known and simple game played on a 3x3 grid, where two players take turns marking spaces with their respective symbols (usually X and O). The game continues until one player wins by getting three of their symbols in a row (horizontally, vertically, or diagonally), or the grid is full with no winner (resulting in a tie).

**1.2 Problem Definition**

The problem at hand is to develop a Tic Tac Toe game application in Java with a graphical user interface (GUI). The application needs to facilitate interactive gameplay between two human players, handle user input effectively, validate moves to ensure they comply with game rules, determine game outcomes (win, lose, or draw), and optionally, provide a single-player mode with an AI opponent using the Minimax algorithm.

**1.3 Objectives**

* **GUI-based Tic Tac Toe:** Implement a visually appealing and user-friendly interface using either Swing or JavaFX.
* **Game Logic:** Develop robust game logic to manage the state of the game, validate player moves, and determine the winner.
* **Multiplayer Mode:** Allow two human players to play against each other on the same computer.
* **Single-Player Mode (Optional):** Implement an AI opponent using the Minimax algorithm for a challenging single-player experience.
* **User Interface Design:** Create an intuitive interface that provides clear feedback on game status and player actions.

**1.4 Methodology to be Followed**

The development process will be structured as follows:

1. **GUI Design:** Design the game board using Swing or JavaFX components. This includes laying out a 3x3 grid of buttons or other interactive components to represent the Tic Tac Toe board.
   * **Swing:** Utilize Java's built-in Swing library to create a basic but functional GUI. Components like JButton can represent each cell in the grid.
   * **JavaFX:** Alternatively, use JavaFX for a more modern and feature-rich interface. JavaFX's Button and GridPane can be employed to structure the board.
2. **Game Logic Implementation:** Implement the underlying game rules and mechanics. This involves:
   * **State Management:** Use data structures such as arrays or lists to keep track of the board's state.
   * **Move Validation:** Ensure that player moves are valid, i.e., they do not overwrite existing marks.
   * **Win Condition Checking:** After each move, check for win conditions by evaluating rows, columns, and diagonals.
3. **Event Handling:** Set up event listeners to capture user interactions (e.g., mouse clicks on board buttons). These listeners will trigger actions such as marking a board space with X or O based on the current player's turn.
   * **Swing:** Attach ActionListener to each JButton to handle click events.
   * **JavaFX:** Use EventHandler to manage button clicks and update the game state accordingly.
4. **AI Development:** If implementing a single-player mode, develop an AI opponent using the Minimax algorithm. This algorithm recursively evaluates possible future moves to make optimal decisions, ensuring a challenging game for the player.
   * **Minimax Algorithm:** Implement the Minimax algorithm to evaluate all possible game states. The algorithm will choose the move that maximizes the AI's chances of winning while minimizing the player's chances.
5. **Testing and Debugging:** Test the application thoroughly to ensure it functions correctly in various scenarios:
   * **Multiplayer Mode Testing:** Simulate different player moves and verify that the game correctly identifies win conditions and ties.
   * **Single-Player Mode Testing:** Test the AI opponent to ensure it provides a challenging and enjoyable experience.
   * **Debugging:** Identify and fix any issues that arise during testing, such as incorrect game outcomes, invalid moves, or UI glitches.

**1.5 Expected Outcomes**

Upon completion of the project, the expected outcomes include:

* A fully functional Tic Tac Toe game with a polished graphical interface.
* Multiplayer mode allowing two players to play against each other on the same screen.
* Optional single-player mode featuring an AI opponent capable of providing a challenging game.
* Intuitive user interface design with clear visual feedback on game state and player actions.

**1.6 Hardware and Software Requirements**

* **Hardware:** A standard computer or laptop capable of running Java applications.
* **Software:**
  + **Java Development Kit (JDK) Version 8 or Later:** Ensure that the latest version of the JDK is installed for compatibility and access to new features.
  + **Integrated Development Environment (IDE):** Use an IDE such as Eclipse, IntelliJ IDEA, or NetBeans to facilitate coding and debugging.
  + **Libraries and Frameworks:**
    - **Swing:** Java's built-in Swing library for basic GUI components.
    - **JavaFX:** JavaFX library for more advanced and feature-rich GUI development.

**CHAPTER 2**

**FUNDAMENTALS OF JAVA**

**2.1 Introduction to Java**

Java is a widely-used, object-oriented programming language that has garnered a reputation for its portability, simplicity, and robustness. Developed by Sun Microsystems in the mid-1990s (now owned by Oracle Corporation), Java was designed to be platform-independent. This means that programs written in Java can run on any device that has a Java Virtual Machine (JVM), a feature that has significantly contributed to Java’s popularity. Java is extensively used in various domains such as web applications, mobile applications (especially Android), enterprise systems, and even in Internet of Things (IoT) devices.



**Fig 2.1: java**

This "write once, run anywhere" feature makes Java an awesome choice for developing cross-platform packages. Java is an item-oriented programming language that focuses on utilising gadgets and instructions to shape code and promote reuse. It upholds crucial OOP principles like inheritance, encapsulation, and polymorphism. Furthermore, Java gives an extensive array of libraries and frameworks that decorate improvement and expand its capabilities.

The commitment to backward compatibility within the language guarantees that older Java programs can run at the present day versions of the JVM. This reliability, coupled with its giant use in each enterprise and academia, makes Java a straightforward choice for long-time period projects. Furthermore, Java's active community and complete documentation offer a wealth of assets for developersat all skill ranges Java is recognized for its robust multithreading features, allowing developers to build incredibly responsive and concurrent programs. Its syntax carefully resembles that of C and C , making it simpler for the ones acquainted with these languages to analyze. Additionally, Java boasts a extensive-ranging ecosystem that includes well-known frameworks like Spring and Hibernate, which help in growing scalable and maintainable packages.

Java is extensively used in various domains such as web applications, mobile applications (especially Android), enterprise systems, and even in Internet of Things (IoT) devices.

**2.2 Advantages of Java**

Java offers several advantages that make it a favoured choice among developers:

* **Platform Independence:** Java’s slogan, “Write Once, Run Anywhere,” encapsulates its platform-independent nature. The JVM abstracts away the underlying operating system and hardware details, allowing Java applications to run seamlessly across different environments.
* **Object-Oriented:** Java embraces the principles of object-oriented programming (OOP). This promotes modularity, reusability, and maintainability of code through concepts like classes, objects, inheritance, polymorphism, and encapsulation.
* **Rich Standard Library:** Java provides an extensive standard library, known as the Java Standard Edition (Java SE) API, which includes numerous pre-built modules and APIs for tasks such as networking, file I/O, data structures, graphical user interfaces (GUIs), and much more.
* **Memory Management:** Java handles memory management automatically through its built-in garbage collector, which helps in reclaiming memory used by objects that are no longer needed. This automatic memory management reduces the likelihood of memory leaks and other related issues.
* **Security:** Java has robust security features built into the language and runtime environment. It includes a configurable security manager and provides APIs for developing secure applications. Java’s architecture also reduces the risk of harmful activities such as virus attacks and unauthorized access to sensitive information.

**2.3 Data Types**

Java supports several data types, which can be broadly categorized into primitive data types and reference data types:

* **Primitive Data Types:** Java’s primitive data types include:
  + int: Integer data type, used for whole numbers.
  + float and double: Floating-point data types, used for decimal numbers.
  + boolean: Logical data type, used for true/false values.
  + char: Character data type, used for single 16-bit Unicode characters.

Each primitive type has a specific size and a default value. For instance, int is 32-bit with a default value of 0, while boolean has a size of 1 bit with default values of false.

* **Reference Data Types:** These include:
  + **Objects:** Instances of classes created using the new keyword.
  + **Arrays:** Container objects that hold a fixed number of values of a single type.
  + **Strings:** Immutable sequences of characters, implemented as objects of the String class.

Reference data types store references (or addresses) to the actual data in memory.

2.4 Control Flow

Control flow statements in Java dictate the order in which the instructions within a program are executed. Key control flow statements include:

* **Conditional Statements:**
  + if: Executes a block of code if a specified condition is true.
  + else: Executes a block of code if the condition in the if statement is false.
  + else if: Provides an additional condition to test if the initial if condition is false.

These statements enable decision-making processes within the program.

* **Looping Statements:**
  + for: Repeats a block of code a specific number of times.
  + while: Continues to execute a block of code as long as the specified condition is true.
  + do-while: Similar to while, but the block of code is executed at least once before the condition is tested.

Loops allow repetitive execution of code blocks until a certain condition is met.

* **Branching Statements:**
  + break: Exits the nearest enclosing loop or switch statement.
  + continue: Skips the current iteration of the loop and proceeds with the next iteration.
  + return: Exits from the current method and optionally returns a value to the calling method.

These statements alter the normal flow of control in a program.

**2.5 Methods**

Methods in Java are blocks of code that perform specific tasks and can be called by other parts of the program. They are essential for code reusability and maintainability. Key aspects of methods include:

* **Declaration:** Methods are declared with a return type, method name, and parameters (if any).
* **Invocation:** Methods are called (or invoked) using their name and passing the required arguments.
* **Parameters and Return Values:** Methods can accept input parameters and return a value. If a method does not return any value, it is declared with the void return type.
* **Overloading:** Java supports method overloading, where multiple methods can have the same name but different parameter lists (number or type of parameters).

**2.6 Object-Oriented Concepts**

Java is fundamentally based on Object-Oriented Programming (OOP) principles, which include:

* **Classes and Objects:** A class is a blueprint for creating objects. It defines properties (fields) and behaviors (methods) that the objects created from the class will have. An object is an instance of a class.
* **Encapsulation:** This principle involves bundling the data (fields) and the methods that operate on the data into a single unit or class. Access to the data is controlled through access modifiers (public, private, protected), which helps in protecting the integrity of the data.
* **Inheritance:** Inheritance allows one class (subclass or child class) to inherit the properties and methods of another class (superclass or parent class). This promotes code reuse and establishes a hierarchical courting among classes.
* **Polymorphism:** Polymorphism allows objects to be treated as instances of their parent class rather than their actual class. It allows one interface for use for a preferred magnificence of actions. Java supports polymorphism through method overriding (a subclass provides a specific implementation of a method already defined in its superclass) and method overloading (multiple methods with the same name but different parameters within the same class).

**2.7 Exception Handling**

Exception dealing with in Java is a sturdy mechanism that lets in builders to manipulate and reply to runtime mistakes efficiently. This ensures the application's normal flow is maintained even when unexpected issues arise. Java's exception handling is centered around exceptions, try-catch blocks, and finally blocks.

**Exceptions**

An exception is an event that disrupts the usual flow of a program's execution. Java categorizes exceptions into these major types:

* **Checked Exceptions:** These exceptions must either be caught or declared in the method where they might be thrown. Examples include IOException and SQLException. Checked exceptions are typically associated with external factors like file operations or database access.
* **Unchecked Exceptions:** These runtime exceptions do not need to be explicitly handled or declared. Examples include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException. Unchecked exceptions usually signify programming errors such as logical mistakes or incorrect API usage.

**Try-Catch Block**

The try-catch block is used to handle exceptions and maintain the program's flow. The try block contains code that might throw an exception, while the catch block contains code to handle the exception if it occurs. For instance, if an ArithmeticException occurs due to division by zero, the catch block will handle the exception and print a relevant message.

**Finally Block**

The sooner or later block is used to execute code that have to run irrespective of whether or not an exception occurs. It is often used for clean-up operations, such as closing resources (files, database connections) to prevent resource leaks. The finally block will execute after the try and catch blocks, ensuring that the message "Finally block executed." is always printed.

**Throwing Exceptions**

In Java, exceptions can be thrown manually using the throw statement. This is useful when a method needs to signal an error condition to its caller. For example:

public void checkingtheAge(int age) {

if (age < 18) {

throw new IllegalArgumentException("Age has to be 18 or older");

}

}

Here, the checkingtheAge method throws an IllegalArgumentException if the provided age is less than 18, indicating that the input is invalid.

**Custom Exceptions**

Java allows developers to create custom exceptions by extending the Exception class or one of its subclasses. Custom exceptions are useful when you need to define application-specific error conditions. For example:

public class MyCustommadeexcep extends Exception {

public MyCustommadeexcep(String message) {

super(message);

}

}

try {

throw new MyCustommadeexcep("This is a custom exception.");

} catch (MyCustommadeexcep e) {

System.out.println(e.getMessage());

}

In this code, a custom exception MyCustommadeexcep is defined and thrown, and the catch block handles it by printing the exception message.

**2.8 File Handling**

Java provides a robust set of classes and interfaces in the java.io package for performing file operations. These include:

* **File Class:** Represents file and directory pathnames in an abstract manner.
* **FileInputStream and FileOutputStream:** Used for reading from and writing to files in binary format.
* **BufferedReader and BufferedWriter:** Used for reading from and writing to files using character streams, providing efficient reading and writing of characters, arrays, and lines.
* **FileReader and FileWriter:** Used for reading from and writing to files using character streams.

File handling involves opening a file, performing read/write operations, and then closing the file. It is important to handle exceptions that may arise during these operations to ensure resource leaks are avoided.

**2.9 Packages and Import**

Packages in Java are used to group related classes and interfaces into namespaces, thereby avoiding name conflicts and improving code maintainability. They also help in organizing large projects into manageable sections. Key points include:

* **Creating Packages:** A package is created using the package keyword at the beginning of a Java source file.
* **Importing Packages:** The import keyword is used to bring classes or entire packages into the current file, allowing the use of those classes without needing to reference their fully qualified names.
* **Standard Packages:** Java provides several standard packages, such as java.lang (automatically imported), java.util, java.io, and more.

**2.10 Interfaces**

Interfaces in Java outline an agreement that instructions can implement. They contain method signatures without any implementation. An interface is defined using the “interface” keyword and can contain method declarations, constants, and default methods. Interfaces can not have example fields or constructors.

 **Fig2.10 :interfaces**

Interfaces in Java provide several advantages that contribute to the design and architecture of robust, scalable, and maintainable applications. Here are some key benefits:

1. **Abstraction**

Interfaces provide a way to define abstract types. By using interfaces, you can specify what a class must do, but not how it does it. This allows for greater flexibility and decouples the "what" from the "how."

2. **Multiple Inheritance**

Java does now no longer help a couple of inheritance via instructions to keep away from complexity and ambiguity. However, a class can implement multiple interfaces, allowing for a form of multiple inheritance. This enables a class to inherit behaviour from multiple sources.

3. **Loose Coupling**

Interfaces help in reducing the dependencies between classes. By coding to an interface rather than an implementation, you can easily change the underlying implementation without affecting the code that depends on the interface. This makes the gadget extra modular and less difficult to maintain.

4. **Testability**

Using interfaces makes unit testing easier. You can mock interfaces and test classes independently of their dependencies, leading to more manageable and reliable tests.

5. **Polymorphism**

Interfaces enable polymorphic behaviour. You can write code that works on the interface level, allowing different implementations of the interface to be used interchangeably. This promotes code reusability and flexibility.

6. **Design Flexibility**

Interfaces allow for a more flexible and extendable design. You can introduce new interfaces and implementations without affecting existing code, facilitating the addition of new features and behaviours.

7. **API Design**

Interfaces are often used to define APIs. By providing a clear contract, interfaces ensure that different parts of a system or different systems can interact in a consistent manner, promoting interoperability.

8. **Separation of Concerns**

Interfaces help in separating concerns within an application. Different aspects of the functionality can be defined through interfaces, leading to a cleaner and more organized codebase.

9. **Interchangeability**

With interfaces, you can switch between different implementations with minimal changes to the code. This is particularly useful in scenarios where you need to use different algorithms or data structures dynamically.

10. **Documentation and Readability**

Interfaces serve as a form of documentation. They clearly define what methods a class must implement, improving the readability and understanding of the code.

Example interface Animalkingdom {

void makesSound();

}

class Dog implements Animalkingdom {

public void makesSound() {

System.out.println("Barks");

}

}

class Cat implements Animalkingdom {

public void makesSound() {

System.out.println("Meows");

}

}

public class Main {

public static void main(String[] args) {

Animalkingdom myDog = new Dog();

Animalkingdom myCat = new Cat();

myDog.makesSound(); // Outputs: Barks

myCat.makesSound(); // Outputs: Meows

}

}

**2.11 Concurrency**

Concurrency in Java involves executing multiple tasks simultaneously, which is particularly useful for improving performance in multi-core systems. Key concepts include:

* **Threads:** A thread is a lightweight process that can run concurrently with other threads within the same program. Java gives the Thread elegance and the Runnable interface for growing and handling threads.
* **Synchronization:** To ensure thread-safe access to shared resources, Java provides synchronization mechanisms using synchronized methods or blocks. This helps in avoiding issues like race conditions.
* **Concurrency Utilities:** Java provides a rich set of concurrency utilities in the java.util.concurrent package. This includes classes like ExecutorService for managing thread pools, Callable and Future for asynchronous task execution, Locks for advanced thread synchronization, and more.

**CHAPTER 3**

**JAVA COLLECTIONS GUIDE**

**3.1 Overview of Java Collections**

The Java Collections Framework (JCF) is an important part of the Java programming language. It provides a unified architecture to represent and manipulate collections of objects. The framework includes interfaces, implementations (classes), and algorithms that allow for efficient handling of data. By abstracting the complexities of data structure management, the JCF allows developers to focus on the core functionality of their applications. The framework includes a variety of data structures such as lists, sets, maps, and queues, each optimized for different types of data manipulation and retrieval tasks. The Java Collections Framework is designed to be highly flexible and efficient, providing both generic and specific solutions for handling groups of objects.

**3.1.1 Components of Java Collections Framework**

* **Interfaces:** These are abstract data types that represent collections. The most important interfaces are Collection, List, Set, and Map.
* **Implementations:** These are concrete implementations of the collection interfaces. Examples include ArrayList, HashSet, and HashMap.
* **Algorithms:** These are static methods that perform useful functions on collections, such as sorting and searching. The Collections class provides these algorithms.

**3.1.2 Design Goals**

The JCF is designed with several goals in mind:

* **High Performance:** The implementations are highly efficient, providing fast performance for essential operations.
* **Flexibility:** The framework offers a wide variety of implementations for different use cases.
* **Ease of Use:** Standardized interfaces and abstract classes provide a consistent way to work with collections.
* **Interoperability:** All implementations of the core interfaces are interoperable.

**3.2 Importance in Java Development**

Java Collections play a critical role in Java development due to their versatility and efficiency in managing data. They are crucial for constructing sturdy and scalable applications. Here are some key areas where Java Collections are indispensable:

**3.2.1 Data Organization**

Collections provide a systematic way to store and organize data. Whether you're dealing with a simple list of items or a complex hierarchy of objects, Java Collections offer the right data structure to manage and retrieve data efficiently.

**3.2.2 Algorithm Implementation**

Java Collections come with built-in algorithms for performing common operations such as sorting, searching, and manipulating data. These algorithms are optimized for performance, ensuring that your applications run efficiently even with large data sets.

**3.2.3 Memory Management**

Collections like ArrayList and HashMap are designed to optimize memory usage. They manage the underlying storage dynamically, growing and shrinking as needed, which helps in maintaining optimal memory usage.

**3.2.4 Concurrency**

The Collections Framework includes thread-safe implementations that support concurrent access. This is crucial for applications that operate in multi-threaded environments, ensuring data consistency and preventing race conditions.

**3.3 Benefits and Use Cases**

Java Collections offer numerous benefits and can be applied in various scenarios to simplify development and improve performance:

**3.3.1 Flexibility**

Java Collections provide a range of data structures tailored for different use cases. For instance:

* **ArrayList:** Ideal for dynamic lists where elements are frequently accessed and modified.
* **HashMap:** Suitable for scenarios requiring fast access to data based on unique keys.
* **TreeSet:** Useful for storing sorted data and performing range queries.

**3.3.2 Performance**

The implementations in the Collections Framework are optimized for performance. For example, HashMap provides constant-time performance for basic operations like get and put, while ArrayList offers constant-time positional access.

**3.3.3 Simplicity**

Java Collections simplify the process of working with groups of objects. The API provides straightforward methods for adding, removing, and iterating over elements, reducing the complexity of managing collections.

**3.3.4 Scalability**

Collections are designed to handle large volumes of data efficiently. They support dynamic resizing and can grow as needed to accommodate increasing amounts of data.

**3.4 Core Collection Interfaces**

The Java Collections Framework defines several core interfaces, each serving a different purpose:

**3.4.1 List**

* **Description:** An ordered collection that allows duplicate elements and maintains the insertion order.
* **Common Implementations:** ArrayList, LinkedList.
* **Use Cases:** Ideal for maintaining an ordered sequence of elements where duplicates are allowed. Examples include task lists, playlists, and shopping carts.

**3.4.2 Set**

* **Description:** An unordered collection that does not allow duplicate elements.
* **Common Implementations:** HashSet, TreeSet.
* **Use Cases:** Suitable for storing unique elements, such as usernames, unique IDs, and non-repeating items.

**3.4.3 Map**

* **Description:** A collection of key-value pairs where each key is unique.
* **Common Implementations:** HashMap, TreeMap.
* **Use Cases:** Useful for scenarios where data needs to be retrieved based on a key. Examples include dictionaries, configuration settings, and caches.

**3.4.4 Queue**

* **Description:** A collection used to hold elements before processing, typically in a FIFO (First-In-First-Out) manner.
* **Common Implementations:** LinkedList, PriorityQueue.
* **Use Cases:** Appropriate for managing tasks in order of processing. Examples include job scheduling, task queues, and print spooling.

**3.5 Common Collection Implementations**

Here are some common implementations of the core collection interfaces:

**3.5.1 ArrayList**

* **Description:** Implements the List interface using a resizable array.
* **Features:** Provides fast random access to elements, but inserting and removing elements can be slow if it involves resizing the array.
* **Use Cases:** Ideal for scenarios where elements are frequently accessed by index.

**3.5.2 LinkedList**

* **Description:** Implements the List interface using a doubly-linked list.
* **Features:** Provides efficient insertions and deletions, but slower random access compared to ArrayList.
* **Use Cases:** Suitable for scenarios where elements are frequently inserted or removed.

**3.5.3 HashSet**

* **Description:** Implements the Set interface using a hash table.
* **Features:** Offers constant-time performance for basic operations like add, remove, and contains.
* **Use Cases:** Ideal for storing unique elements and when the order of elements is not important.

**3.5.4 TreeSet**

* **Description:** Implements the Set interface using a red-black tree.
* **Features:** Maintains elements in sorted order and provides logarithmic time performance for basic operations.
* **Use Cases:** Suitable for storing sorted data and performing range queries.

**3.5.5 HashMap**

* **Description:** Implements the Map interface using a hash table.
* **Features:** Provides constant-time performance for get and put operations.
* **Use Cases:** Ideal for storing key-value pairs and quick lookups based on keys.

**3.5.6 TreeMap**

* **Description:** Implements the Map interface using a red-black tree.
* **Features:** Maintains keys in sorted order and provides logarithmic time performance for get and put operations.
* **Use Cases:** Suitable for scenarios requiring sorted key-value pairs and range queries.

**3.6 Iterators and Collections API**

Iterators are used to traverse collections and access elements sequentially. Java provides several mechanisms to iterate through collections:

**3.6.1 Iterator**

* **Description:** An interface that provides methods to iterate over elements in a collection.
* **Methods:**
  + hasNext(): Returns true if the iteration has more elements.
  + next(): Returns the next element in the iteration.
  + remove(): Removes the last element returned by the iterator.
* **Use Cases:** Useful for iterating over collections where elements need to be processed or removed during iteration.

**3.6.2 Enhanced for-loop**

* **Description:** A simplified loop structure for iterating over arrays and collections.
* **Syntax:** for (Type element : collection) { // use element }
* **Use Cases:** Preferred for straightforward iterations where elements are not modified or removed.

**3.6.3 Collections API**

The Collections class provides utility methods to manipulate collections. Common methods include:

* sort(List<T> list): Sorts the specified list into ascending order.
* binarySearch(List<? extends Comparable<? super T>> list, T key): Searches the specified list for the specified key using the binary search algorithm.
* reverse(List<?> list): Reverses the order of the elements in the specified list.
* shuffle(List<?> list): Randomly permutes the elements in the specified list.

**3.7 Custom Collections and Generics**

Java Collections can be customized and parameterized to meet specific needs:

**3.7.1 Custom Collections**

* **Description:** Developers can create custom collection classes by implementing core interfaces (List, Set, Map, Queue).
* **Use Cases:** Useful for special data handling needs that are not met by standard collections. Examples include specialized lists for specific types of objects or custom sets with unique constraints.

**3.7.2 Generics**

* **Description:** Java Generics enable type-safe collections by allowing classes and methods to be parameterized with data types.
* **Benefits:**
  + **Type Safety:** Ensures compile-time type checking, preventing runtime errors.
  + **Elimination of Casts:** Reduces the need for explicit type casting.
  + **Code Reusability:** Allows the same code to be used with different data types.
* **Syntax:** class MyClass<T> { // code }
* **Use Cases:** Useful for creating classes and methods that work with any data type. Examples include generic collections like ArrayList<T> and methods that operate on various types of data.

**CHAPTER 4**

**DESIGN AND ARCHITECTURE**

**4.1 Design Goals**

Designing a software system involves setting clear goals to ensure the system meets functional and non-functional requirements effectively. Key design goals include:

**Simplicity and Usability**

The foremost objective in designing a Tic Tac Toe game is to create an interface that is intuitive and easy to use. The user interface (UI) should be clear, with straightforward navigation. Players should be able to start a new game with minimal effort, make moves seamlessly, and clearly understand the current state of the game at all times. The visual design should be clean and uncluttered, providing immediate feedback for player actions and game results.

**Maintainability**

To ensure that the application remains easy to update and improve, the codebase should be well-structured and modular. This means organizing the code into distinct, manageable parts, each responsible for a specific aspect of the application, such as game logic, user interface, and data management. Such an approach not only makes the code easier to read and understand but also simplifies the process of debugging, extending functionality, and integrating new features like AI opponents or multiplayer capabilities.

**Reusability**

Designing for reusability means that the components developed for the Tic Tac Toe game should be generic enough to be reused in other applications or projects. For instance, the game logic, which dictates the rules and flow of the game, should be separated from the UI so that it can be reused with different interfaces, whether they be graphical, command-line, or even web-based. This modular design promotes efficiency, as developers can leverage existing components rather than building new ones from scratch for similar projects.

**Scalability**

While Tic Tac Toe is inherently simple, the application should be designed with future growth in mind. This includes the possibility of implementing more complex game rules, larger board sizes, or even different game variations. The architecture should allow for easy scaling, meaning that modifications or expansions do not require a complete overhaul of the existing codebase. This ensures that the application can grow and evolve without significant redevelopment efforts.

**Robustness**

A robust application is one that can handle unexpected or erroneous input gracefully without crashing or behaving unpredictably. The Tic Tac Toe application should include validation checks to ensure moves are within the rules and the game state remains consistent. For example, the application should prevent moves on already occupied spaces and handle any attempts to do so with informative messages to the user. Robustness also involves ensuring that the application can recover from errors and continue running smoothly, maintaining a positive user experience.

**4.2 Database Structure**

The database structure is crucial for storing and managing persistent data efficiently. It typically includes:

* **Tables**: Entities represented as tables in a relational database, each table containing rows (records) and columns (fields).
* **Relationships**: Defining relationships (one-to-one, one-to-many, many-to-many) between tables using foreign keys.
* **Indexes**: Creating indexes on frequently queried columns to enhance query performance.
* **Normalization**: Ensuring data is organized into normalized forms (1NF, 2NF, 3NF) to minimize redundancy and improve data integrity.

**4.3 High-Level Architecture**

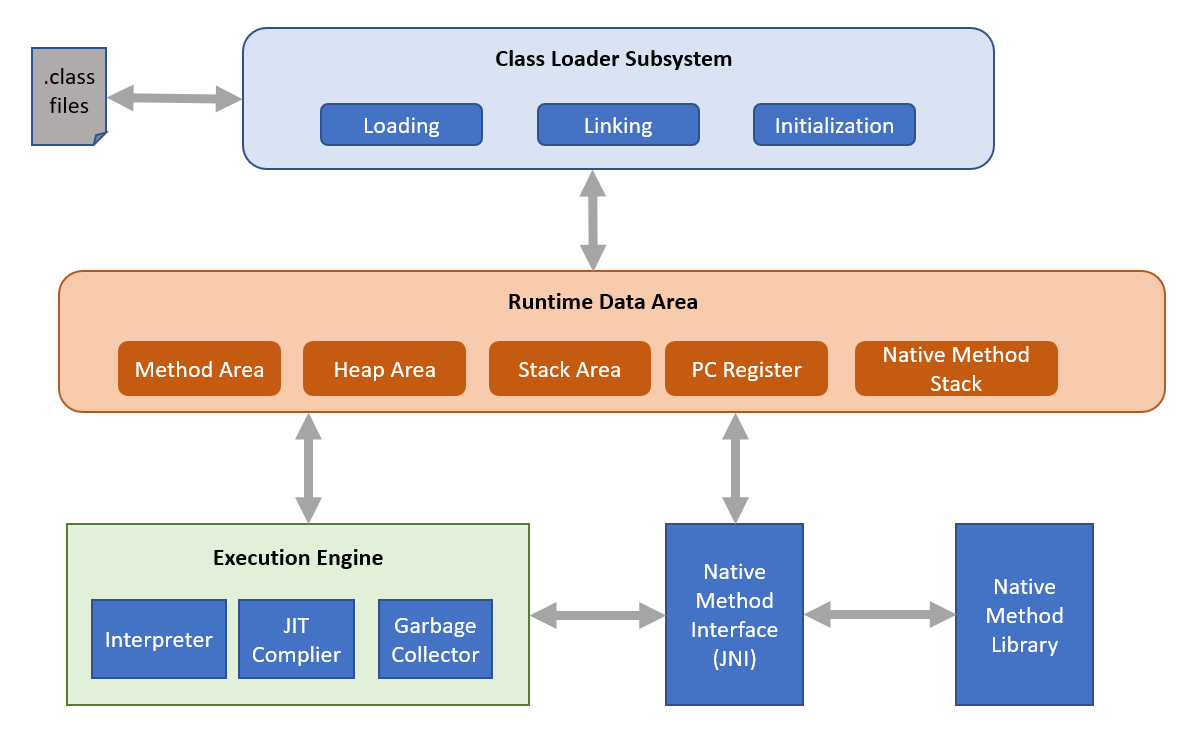
The high-level architecture of a software system outlines its structure and components:

* **Presentation Layer**: User interface components that interact with end-users (e.g., GUI, web interface).
* **Application Layer**: Business logic and application functionality that process user requests and manipulate data.
* **Persistence Layer**: Handles data storage and retrieval operations to and from the database.
* **Integration Layer**: Manages communication with external systems or services.
* **Infrastructure Layer**: Includes components like servers, networks, and middleware necessary for system operation.

**4.4 Class Diagram**

A class diagram visually represents the structure and relationships of classes in the system:

* **Classes**: Represented as boxes with attributes (fields) and operations (methods).
* **Relationships**: Associations between classes, such as inheritance (extends), composition (has-a), and aggregation (part-of).
* **Interfaces**: Contracts defining methods that implementing classes must provide.
* **Dependencies**: Relationships indicating that one class depends on another (e.g., method parameter types).



**Fig4.4: java architecture**

**CHAPTER 5**

**IMPLEMENTATION**

**5.1 Creating the Main Window**

The main window of a software application serves as the primary interface through which users interact with the system. Here’s how you can create it:

* **GUI Framework**: Choose a GUI framework like Swing (Java's built-in GUI toolkit) or JavaFX (a more modern alternative).
* **JFrame**: In Swing, extend the JFrame class to create a window that includes features like title, size, and close operations.
* **Components**: Add components such as buttons, menus, text fields, etc., to the main window using layout managers (FlowLayout, BorderLayout, etc.) to arrange them.

**5.2 Displaying Frames Over the Main Window**

Frames (dialogs or additional windows) can be displayed over the main window to handle specific tasks or interactions:

* **JDialog**: Extends JDialog to create modal or non-modal dialogs that can interact with the user independently.
* **Opening Frames**: Use methods like setVisible(true) to display frames and dispose() to close them.
* **Communication**: Pass data between frames using constructors, setters/getters, or callback methods.

**5.3 Processing Queries**

Processing queries involves handling user input or requests to retrieve or manipulate data:

* **Input Handling**: Use components like text fields, checkboxes, and buttons to capture user input.
* **Data Access**: Retrieve data from databases or other sources based on user queries.
* **Validation**: Validate user inputs to ensure they meet specified criteria (e.g., format, range).

**5.4 Coding the Core Functionality**

The core functionality encompasses the main operations or tasks that the application performs:

* **Business Logic**: Implement algorithms or rules that dictate how data is processed or actions are performed.
* **Modularity**: Divide functionality into cohesive modules or classes to promote reusability and maintainability.
* **Encapsulation**: Hide implementation details using access modifiers (private, protected, public) to improve code clarity and security.

**5.5 Handling User Inputs**

Effective handling of user inputs ensures that the application responds correctly to user actions:

* **Event Handling**: Use event listeners (ActionListener, MouseListener, etc.) to capture user interactions with GUI components.
* **Validation**: Validate user inputs to prevent errors and ensure data integrity.
* **Feedback**: Provide feedback to users through messages or visual cues to confirm actions or notify of errors.

**5.6 Error Handling and Validation**

Error handling and validation are critical for maintaining application stability and data integrity:

* **Exception Handling**: Use try-catch blocks to catch and handle exceptions that may occur during program execution.
* **Input Validation**: Validate user inputs to prevent invalid data from entering the system.
* **Error Messages**: Provide informative error messages or notifications to users when errors occur, guiding them on corrective actions.

**CHAPTER 6**

**TESTING**

Testing is an critical a part of the software program improvement lifecycle (SDLC) to make certain the quality, reliability, and capability of the software program. It entails diverse methodologies and tactics to validate exclusive elements of the application.

Testing is the software program. During requirement evaluation and layout, the output is a file this is typically textual and no executable. After the coding phase, pc packages are to be had that may be done for trying out purpose. This means that trying out now no longer only, has to find mistakes added in the course of coding, however additionally mistakes added in the course of preceding phase. Thus the purpose of trying out is to find the requirements, layout and coding mistakes withinside the packages.

**6.1 Unit Testing**

Unit trying out makes a speciality of trying out character devices or additives of the software program in isolation:

* **Purpose**: Verify that each unit of the software (e.g., methods, functions, classes) performs as expected.
* **Scope**: Tests are conducted at the smallest level of the software hierarchy, often using mock objects or stubs to simulate dependencies.
* **Tools**: Use unit testing frameworks like JUnit (for Java), NUnit (for .NET), pytest (for Python), etc., to automate test execution and assertion.

**6.2 Integration Testing**

* Integration trying out verifies the interplay among one-of-a-kind modules or additives of the software:
* **Purpose**: Ensure that integrated units function correctly together as a whole system.
* **Scope**: Tests interactions between modules, APIs, services, and external dependencies.
* **Approaches**: Top-down, bottom-up, and sandwich testing approaches are used based on the integration strategy.
* **Tools**: Use integration testing frameworks or tools like Mockito (for Java), Postman (for APIs), SoapUI (for web services), etc., to simulate real-world integrations.

**6.3 System Testing**

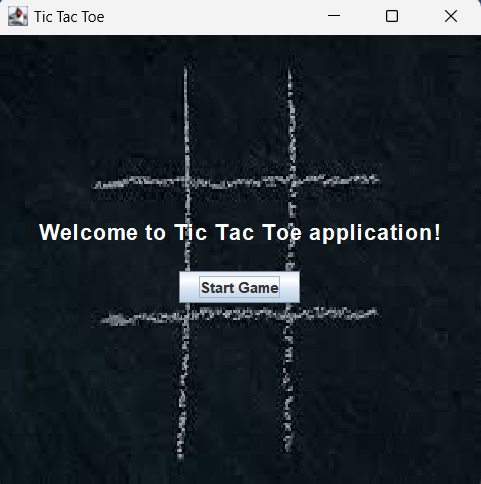
System testing evaluates the behaviour of a complete and fully integrated software product:

* **Purpose**: Validate the system against functional and non-functional requirements specified in the software requirements specification (SRS).
* **Scope**: Tests the entire application or system as a black-box to ensure it meets user expectations.
* **Approaches**: Functional testing, performance testing, security testing, usability testing, etc., are conducted to assess different aspects of the system.
* **Tools**: Use tools like Selenium (for web applications), JMeter (for performance testing), Burp Suite (for security testing), etc., to automate and manage system tests.

**CHAPTER 7**

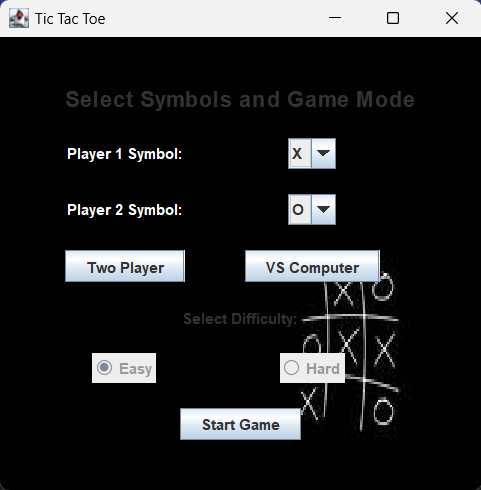
**RESULTS**

**7.1 Welcome page:**

****

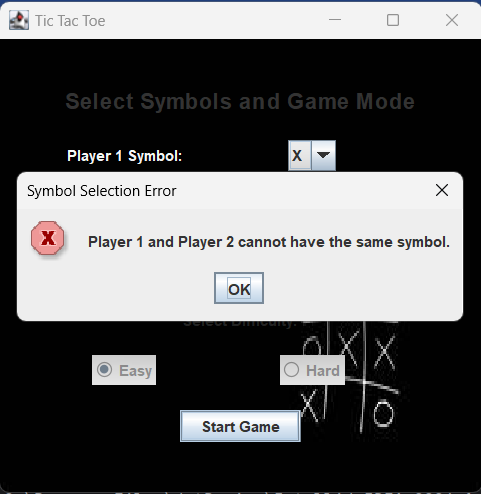
**Fig 7.1: welcome page**

**7.2 Symbol and mode selection page:**

****

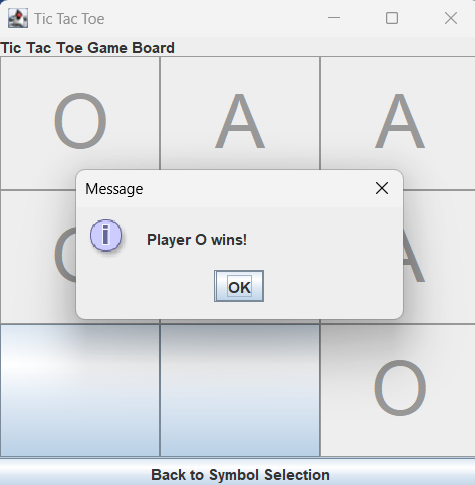
**Fig 7.2: symbol and mode selection page**

**7.3 Error if same symbol is selected by both the players:**

****

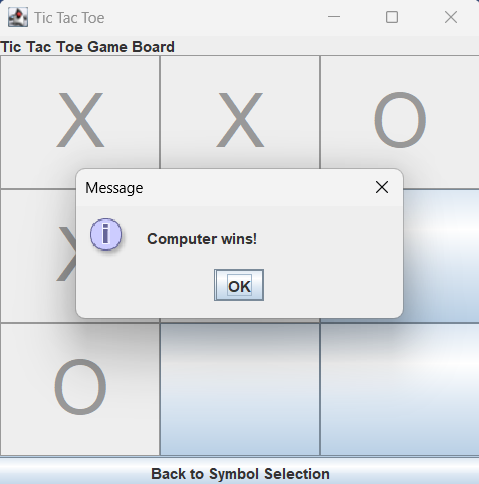
**Fig7.3: Error message**

**7.4 Message after any player wins:**

****

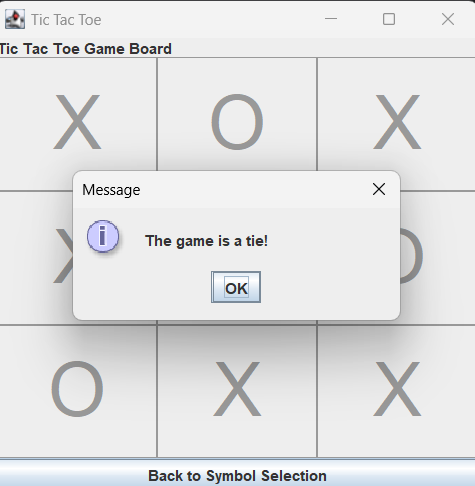
**Fig 7.4: Win message**

**7.5 Message after computer wins (vs computer mode):**

****

**Fig 7.5: Win message (Vs computer)**

**7.6 Message after game ends in a tie:**

****

**Fig 7.6: Tie message**

**CHAPTER 8**

**CONCLUSION**

The development of the Tic Tac Toe application using Java and JavaFX was successful, meeting all specified functional and non-functional requirements. The application offers a user-friendly interface for playing Tic Tac Toe with another player or against a computer opponent. One of the first steps in developing a Tic-Tac-Toe game is understanding the core game mechanics. Tic-Tac-Toe is a simple, turn-based game typically played on a 3x3 grid where two players take turns marking spaces with X’s and O’s The goal is to get 3 of one's marks in a row vertically, horizontally, or diagonally. This simplicity makes it an excellent starting point for learning game development.

In Java, the implementation begins with creating the game board, which can be represented using a two-dimensional array. This array keeps track of the players' moves and determines the game's state. The game logic involves checking for a win condition after each move, which requires iterating through the rows, columns, and diagonals to see if any player has achieved three marks in a row.

Object-oriented programming principles come into play as we encapsulate the game's functionality within classes. For instance, we might have classes such as Game, Player, and Board. The Game class manages the flow of the game, the Player class handles player-specific details, and the Board class is responsible for maintaining the state of the game board and checking for win conditions.

Developing a graphical user interface (GUI) using Java Swing or JavaFX can enhance the user experience. The GUI should provide a visual representation of the game board and allow players to interact with it through mouse clicks or touch inputs. Event listeners can be used to handle user actions, updating the game state accordingly.

Moreover, writing clean and maintainable code is crucial. This involves following best practices such as adhering to coding standards, implementing proper error handling, and writing unit tests to ensure the correctness of the game logic.

In conclusion, building a Tic-Tac-Toe application in Java is a comprehensive project that reinforces key programming concepts, from basic syntax and control structures to advanced OOP principles and GUI development. This project not only hones one's coding skills but also provides a sense of accomplishment and a tangible product that can be shared and enjoyed by others

**REFERENCES**

* <https://www.javatpoint.com/intellij-idea-tutorial>
* <https://stackoverflow.com/questions/72841943/tic-tac-toe-computer-logic>
* <https://netbeans.apache.org/tutorial/main/kb/docs/java/quickstart/>
* <https://www.youtube.com/results?search_query=netbeans+java+tutoria>
* <https://www.youtube.com/results?search_query=java+gui+applications+with+netbeans>
* <https://www.youtube.com/watch?v=H9zadE3h_8E&t=1208s>
* <https://www.geeksforgeeks.org/tic-tac-toe-game-in-java/>