Piece representation:

```
package ticTacToe;

/** Representation of X's and O's in each square. */
enum Piece {
    X, O, EMP;

    Piece opposite() {
        switch (this) {
            case X:
                return O;
            case O:
                return X;
            default:
                return null;
        }
    }
}
```

Board representation:

```
package ticTacToe;
import java.util.ArrayList;
import java.util.Arrays;
import static ticTacToe.Piece.*;

/** Representation of 3-by-3 board and placement of X's and O's. */
public class Board {
    Piece[] state;
    Piece turn;

    public Board() {
        state = new Piece[9];
        for (int i = 0; i < 9; i++) {
            state[i] = EMP;
        }
        turn = X;
    }

    /** Copy constructor. Creates new Board with same state as B. */
    public Board(Board b) {
        state = new Piece[9];
        System.arraycopy(b.state, 0, state, 0, 9);
        turn = b.turn;
    }

    /** Returns copy of board with move made at position POS */
    public Board makeMoveCopy(int pos) {
        assert pos >= 0 && pos < 9;
        Board copy = new Board(this);
        copy.state[pos] = turn;
        copy.turn = turn.opposite();
        return copy;
    }
```

```
    /** Returns winner, or EMP if tie, and null if no winner.
      * Possible win sequences: 012, 345, 678, 036, 147, 258, 048, 246 */
    public Piece winner() {
        Piece a = state[0], b = state[1], c = state[2];
        Piece d = state[3], e = state[4], f = state[5];
        Piece g = state[6], h = state[7], i = state[8];
        if (a != EMP && ((a == b && a == c) || (a == d && a == g)
                || (a == e && a == i))) {
            return a;
        } else if (b != EMP && (b == e && b == h)) {
            return b;
        } else if (c != EMP && ((c == f && c == i)
                || (c == e && c == g))) {
            return c;
        } else if (d != EMP && (d == e && d == f)) {
            return d;
        } else if (g != EMP && (g == h && g == i)) {
            return g;
        } else if (empties().length == 0) {
            return EMP;
        }
        return null;
    }

    /** Returns array containing indices of empty squares. */
    public int[] empties() {
        ArrayList<Integer> arr = new ArrayList<>();
        for (int i = 0; i < state.length; i++) {
            if (state[i] == EMP) {
                arr.add(i);
            }
        }
        int[] res = new int[arr.size()];
        for (int i = 0; i < arr.size(); i++) {
            res[i] = arr.get(i);
        }
        return res;
    }

    @Override
    public boolean equals(Object o) {
        if (o == null || o.getClass() != Board.class) return false;
        return Arrays.equals(state, ((Board) o).state);
    }

    @Override
    public int hashCode() {
        return Arrays.hashCode(state);
    }
}
```

Game representation:

```
package ticTacToe;
import static ticTacToe.Piece.*;

public class Game {

    public static Board doMove(Board board, int move) {
        return board.makeMoveCopy(move);
    }

    public static int[] generateMoves(Board board) {
        if (!primitiveValue(board).equals("not_primitive")) {
            return new int[0];
        }
        return board.empties();
    }

    public static String primitiveValue(Board board) {
        Piece winner = board.winner();
        if (winner == EMP) {
            return "tie";
        } else if (winner != null) {
            return "lose";
        }
        return "not_primitive";
    }
}
```

Solver:

```
package ticTacToe;
import static ticTacToe.Game.*;

import java.util.HashMap;

public class Solver {

    /** Keeps track of already-computed positions. */
    HashMap<Board, String> memo;

    /** Count indices: 0 - LOSE, 1 - primitive LOSE;
      * 2 - WIN, 3 - primitive WIN; 4 - TIE, 5 - primitive TIE*/
    int[] counter;

    public Solver() {
        memo = new HashMap<>();
        counter = new int[6];
    }

    /** Solves BOARD from given position and returns win-value of position.
    public String solve(Board board) {
        if (memo.containsKey(board)) {
            return memo.get(board);
        }
        String pVal = primitiveValue(board);
        if (!pVal.equals("not_primitive")) {
            if (pVal.equals("lose")) {
                counter[1] += 1;
            } else if (pVal.equals("win")) {
                counter[3] += 1; // never happens in this game
            } else {
                counter[5] += 1;
            }
            return memoize(board, pVal);
        }
        int[] moves = generateMoves(board);
        String[] future = new String[moves.length];
        for (int i = 0; i < moves.length; i++) {
            Board newBoard = doMove(board, moves[i]);
            future[i] = solve(newBoard);
        }
        boolean winFlag = true; // T if all children are win positions
        for (String f : future) {
            if (f.equals("lose")) {
                return memoize(board, "win");
            } else {
                winFlag = winFlag && f.equals("win");
            }
        }
        return memoize(board, winFlag ? "lose" : "tie");
    }
```

```
    /** Add board and win-value to MEMO and increment counter. */
    public String memoize(Board b, String value) {
        if (!memo.containsKey(b)) {
            memo.put(b, value);
            switch(value) {
                case "lose":
                    counter[0] += 1;
                    break;
                case "win":
                    counter[2] += 1;
                    break;
                case "tie":
                    counter[4] += 1;
                    break;
            }
        }
        return value;
    }

    public void printCounts() {
        System.out.printf("Lose: %d (%d primitive)\n", counter[0], counter[1]);
        System.out.printf("Win: %d (%d primitive)\n", counter[2], counter[3]);
        System.out.printf("Tie: %d (%d primitive)\n", counter[4], counter[5]);
        System.out.printf("Total: %d (%d primitive)\n\n",
            counter[0] + counter[2] + counter[4],
                counter[1] + counter[3] + counter[5]);
    }

    public static void main(String[] args) {
        Board b = new Board();
        Solver s = new Solver();
        long t = System.currentTimeMillis();
        s.solve(b);
        s.printCounts();
        System.out.println("Time: " + (System.currentTimeMillis() - t) + " ms");
    }

}
```

OUTPUT of Solver.main():

```
Lose: 1574 (942 primitive)

Win: 2836 (0 primitive)

Tie: 1068 (16 primitive)

Total: 5478 (958 primitive)


Time: 106 ms
```