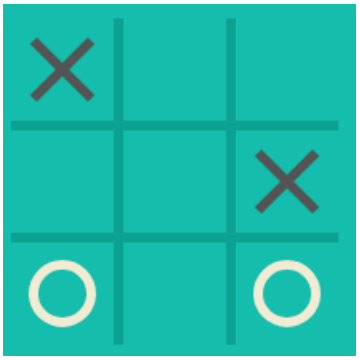
AI Player

Finally we turn our attention to creating the `AIPlayer` class. The goal of this class is to create a computer controlled "player" that will learn to play the game of Tic-Tac-Toe perfectly (i.e. choose moves to play in a game in such way that it will the game if possible, or tie the game.)

To make the program "learn" to play Tic-Tac-Toe, we are going to use a strategy called Reinforcement Learning, or "Q-Learning". The "Q" stands for Quality - that is the program will learn to assign a particular Quality to each possible move, and will choose the move with the best quality each time.

Q-Learning depends on the computer developing a memory of the **STATE** of Tic-Tac-Toe game, and which move (or **ACTION**) it choose when the game was in that state. For every "STATE-ACTION" pair, the algorithm will assign a **QUALITY** to that choice, based on the reward for making that decision.

The **state** of the game will be given as a String representing the game board at a given moment. For example the board:



Would be represented "`x....xo.o`"

The chosen action for a given state will be given by a number representing the square the computer chooses to play in. Each possible square will be numbered using the values 0 through 8.

So the **STATE-ACTION** pair "`x....xo.o-7`" in this case, could represent placing the token ("O") in the second column of the third row of the board.

This should be seen as a high **quality** move (because it wins the game!). A double value will be used to represent the quality of a given move. A move that wins the game may have a quality of 10.0.

Creating and Storing the AI Player's Memory

```java

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Random;

public class AI {

    private final Random RNG = new Random();
    private final double LEARNING_RATE = 0.75;
    private final double DISCOUNT_FACTOR = 0.5;

    /* As it plays, the AI generates a memory of State-Action pairs
       The state represents the current look of the board
       The action is the move the computer chooses in that state
       Each state-action pair is mapped to a double representing the quality of that move
       Over time, the quality of moves is adjusted up or down as the computer learns
       E.g. "X....XO.O-7" -> 5.75
            -the board state is "X....XO.O" and the chosen move is 7.
            -the quality of this move is rated 5.75 */
    HashMap<String, Double> memory = new HashMap<>();
```

Within the code, we will use an object called a HashMap to create this memory.

A HashMap creates a table of values that matches one object (In this case a String representing a STATE-ACTION pair) with a second object (in this case a Double value representing the quality of that action choice).

The computer learns to play to Tic-Tac-Toe by finding and saving around 16,000 different state-action Strings, rating the quality of each, and storing them in this table of values. It takes somewhere around 30,000-50,000 games for the AI to develop an optimal strategy

Choosing a Move

How does the computer decide what move it should choose?

The AI Player needs to look into it's memory to see if it has ever encountered a game board matching the current game before.

If it has never seen a match before, the computer will randomly choose a possible move.

If it has seen a match before, the computer should choose the highest quality move out of all the known moves matching this board.

We also want a certain amount of chance that the computer will EXPLORE new moves rather choosing the best known move. This way the algorithm does not get stuck on choosing a known move when a move it has never tried before might actually be better.

In code, that looks like:

```
24      public int chooseMove(Game game, double exploreProbability) {
25          //first make sure that the game is still going
26          if (game.getStatus() != Game.GAME_CONTINUES) return -1;
27
28          //get a string representing the current board state
29          String state = game.getState();
30
31          //Check the memory to see if we have ever encountered this state before
32          List<Entry<String, Double>> previousActions = memory.entrySet().stream()
33                  .filter(entry -> entry.getKey().startsWith(state))
34                  .toList();
35
36
37          //choose a random move if we have never seen this state before,
38          // or randomly choose to explore
39          if (previousActions.isEmpty() || RNG.nextDouble() < exploreProbability) {
40              List<Integer> moves = game.availableMoves();
41              return moves.get(RNG.nextInt(moves.size()));
42          }
43
44          //if we have seen this state before, look for the best known action
45          double maxQuality = previousActions.stream()
46                  .map(Map.Entry::getValue)
47                  .max(Double::compare)
48                  .orElse(0.0);
49
50          //create a list of all previous actions tied for the highest quality
51          List<String> tiedActions = previousActions.stream()
52                  .filter(entry -> entry.getValue().equals(maxQuality))
53                  .map(Map.Entry::getKey)
54                  .toList();
55
56          //Choose an action off this list
57          String chosenAction = tiedActions.get(RNG.nextInt(tiedActions.size()));
58
59          //Chosen action will be the last digit of the chosenAction, converted to int
60          return Integer.parseInt(chosenAction.substring(chosenAction.length()-1));
61      }
```

AI Learning

The algorithm is programmed to choose the highest "quality" move, but how will the algorithm determine "quality"? Simple, we award points for making choices that lead to the desired state. In Tic-Tac-Toe the most desirable state is winning the game! The second most desirable state is having a tie game. So we can define a reward method for a game of TicTacToe something like:

```java
    /**
     * Calculate the reward for a move given the status of the game after
     * making the move.
     * @param status
     * @return 10.0 for a win, 5.0 for a draw, 0.0 otherwise
     */
    private double getReward(int status) {
        switch(status) {
            case Game.X_WINS:
            case Game.O_WINS:
                return 10.0;
            case Game.DRAW:
                return 5.0;
            default:
                return 0.0;
        }
    }
```

In order to learn to choose better moves, each time the computer makes a move, will update the value of the quality of the move it just made according to the formula

$$newQuality = (learningRate) * (reward - discount * bestNextMove) + (1 - learningRate) * (currentQuality)$$

This formula creates (mathematically) a blend between what the currently remembered quality move of taking this move, and the new information learned by taking this move this time (based on any rewards). The formula also provides a certain value for "future" moves. For example, a move that doesn't win on this turn, but sets the AI up to win on it's next turn, will be credited some "future value". The future value is somewhat discounted however, because the AI may or may not receive this future value.

In code we have:

```java
    private final double LEARNING_RATE = 0.75;
    private final double DISCOUNT_FACTOR = 0.5;
```

```java
    /**
     * Make an update to the quality stored in the memory table based on the reward for a move
     * @param stateAction   - starting state and action chosen
     * @param reward        - value for making this action
     * @param newState      - state after taking this action
     */
    public void updateQualityValue(String stateAction, double reward, String newState) {
        //Check the memory for the known quality of the given state/action string (defaulting as 0.0 if it is unknown)
        double currentQuality = memory.getOrDefault(stateAction, 0.0);

        //find out the quality of the best known "next action"
        double bestNextMove = memory.entrySet().stream()
                .filter(entry -> entry.getKey().startsWith(newState))
                .map(Map.Entry::getValue)
                .max(Double::compare)
                .orElse(0.0);

        double newQuality = currentQuality + LEARNING_RATE * (reward - DISCOUNT_FACTOR * bestNextMove - currentQuality);

        //if there is a change to make, make it
        if (currentQuality != newQuality) {
            memory.put(stateAction, newQuality);
        }
    }
```

(The mathematical formula has be rearranged slightly here, to make a more efficient calculation, but it is algebraically the same).

Training the AI

Finally, now that the AI can choose the best move it knows and can update the value of each move taken, we turn to the task of training the AI to play the game well. The AI will need to learn to play somewhere around 16,000 different moves, and rate the quality of each of these. In order to do that, it will need to play MANY MANY games. Fortunately, we can simply have the AI play against itself!

```
89      /**
90       * Trains the AI by playing games against itself
91       * @param games number of games to play for training
92       */
93      public void train(int games) {
94          //handy to have some type of progress displayed on screen for this
95          //because it can be lengthy
96          System.out.printf("Starting training of %d games.\n", games);
97          System.out.println("---------1---------2---------3---------4---------5");
98
99          //train for the give number of games
100         for (int i = 0; i < games; i++) {
101             playOneGame();
102             if ( i % 1000 == 0) System.out.print("*");
103         }
104
105         //give some idea of how the memory has changed because of this training
106         System.out.println();
107         System.out.println("Training completed.");
108         System.out.println("Memory now contains " + memory.size() + " quality values");
109     }
```

To play a single game against itself we have the following code:

```
111     /**
112      * play one training game, updating rewards memory when the game finishes
113      */
114     private void playOneGame() {
115         Game game = new Game();
116
117         //get the initial state of the game (blank board...)
118         String startingState = game.getState();
119
120         //play the game until a draw or one player wins
121         while (game.getStatus() == Game.GAME_CONTINUES) {
122             //while training, we give the AI at least some chance to choose
123             //random moves so that it explores moves it may not have tried
124             int move = chooseMove(game, 0.50);
125             game.play(move);
126
127             //calculate a reward for this move
128             double reward = getReward(game.getStatus());
129             String newState = game.getState();
130
131             //update the Quality table with results of this move
132             updateQualityValue(startingState + ":" + move, reward, newState);
133
134             //remember this as the starting state for the next move
135             startingState = newState;
136         }
137
138     }
```