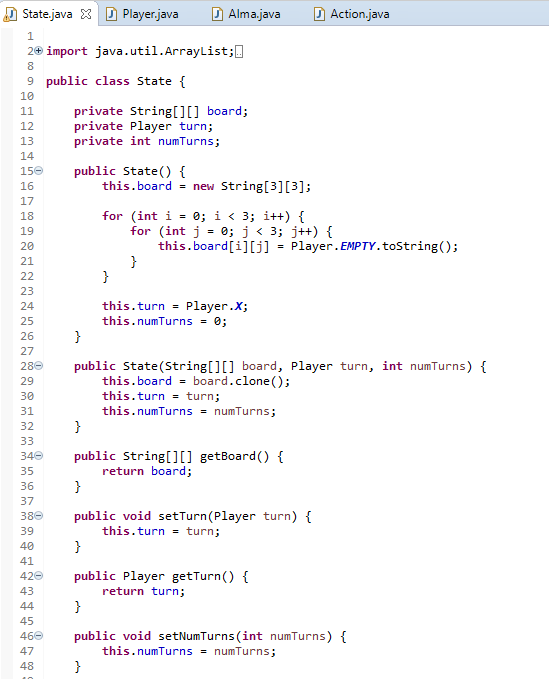
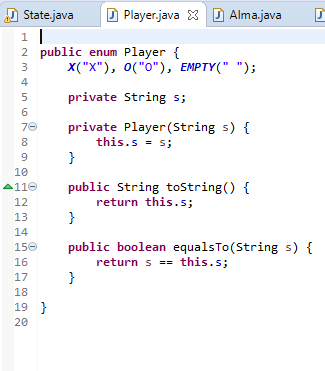
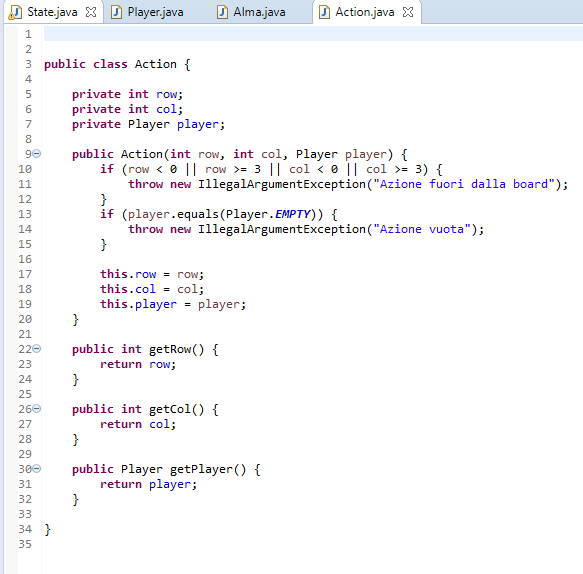
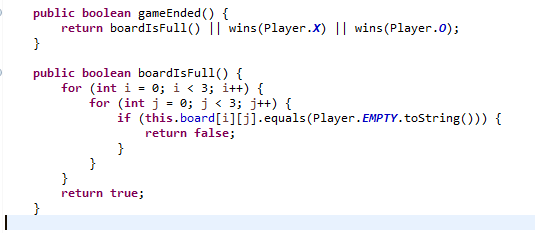
**Una semplice IA che gioca a TRIS**

1)Per creare una IA occorre come prima cosa rappresentare il gioco che si vuole considerare. Nel caso del tris è stato scelta una matrice 3X3 che viene inizializzata nel costruttore a un qualsiasi parametro che indica “libero” o “vuoto” o che rappresenta la board (per esempio negli scacchi occorre settare la posizione iniziale dei pezzi).  


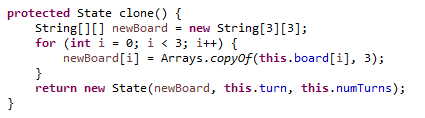


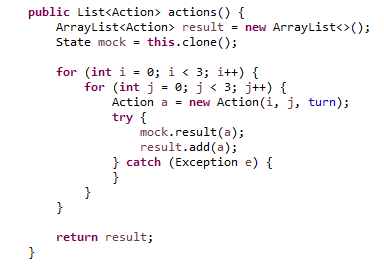
2)Dopo aver creato la board di gioco occorre il concetto di **azione** ovvero la “mossa” che è in grado di modificare lo stato generandone uno diverso.  
l’azione nel caso del tris è semplicemente il “dichiarare” una riga-colonna e, dopo aver verificato che la mossa è legale (non va sopra ad altre azioni o non è una mossa vuota), eseguirla.  
  


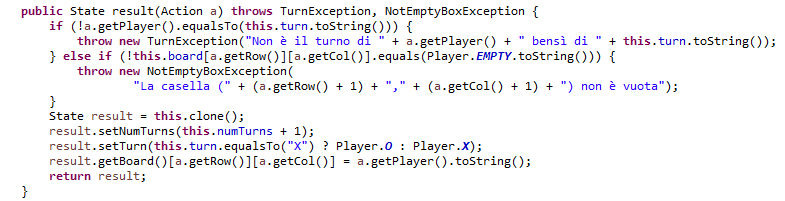
3)Nello stato sono solitamente necessarie alcune funzioni di “utilità” che verificano se il **gioco è terminato** oppure se la **board di gioco è piena** ma soprattutto **chi/come ha vinto.**Nel caso del tris il gioco termina se la board è piena (**pareggio**) oppure se uno dei due giocatori vince (rappresentato dall’enumerativo Player.X o Player.O)  
  


Per verificare se un giocatore ha **vinto** è stata definita la funzione **wins**   
Essa controlla le X o i O consecutivi e verifica se il giocatore corrente ha vinto (il controllo viene fatto se il numero di turni è superiore a 5 per evitare di controllarlo ad ogni turno dato che sarebbe inutile visto che nel tris si può vincere dal 5° turno in poi. (Un turno viene contato come tempo PERSONALE e tempo AVVERSARIO ed è incrementato direttamente nello stato nella variabile privata numTurns)

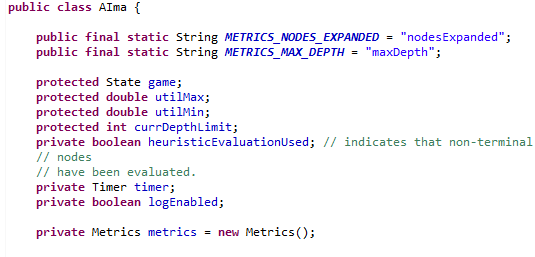
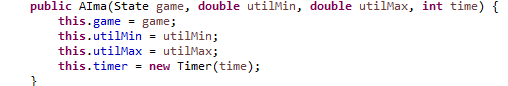
Un’altra funzione molto utile è la clone(). Essa consente di **copiare lo stato corrente** perché a causa dei vari riferimenti si potrebbe distruggere la board di partenza.



4)Per **generare gli stati futuri**, occorre una funzione in grado di restituire **la lista di tutte le possibili azioni effettuabili da un giocatore in un dato turno**. Tale funzione ritorna quindi una **lista di azioni** ognuna delle quali è individuata con coordinate X Y (come definito nel suo costruttore)  


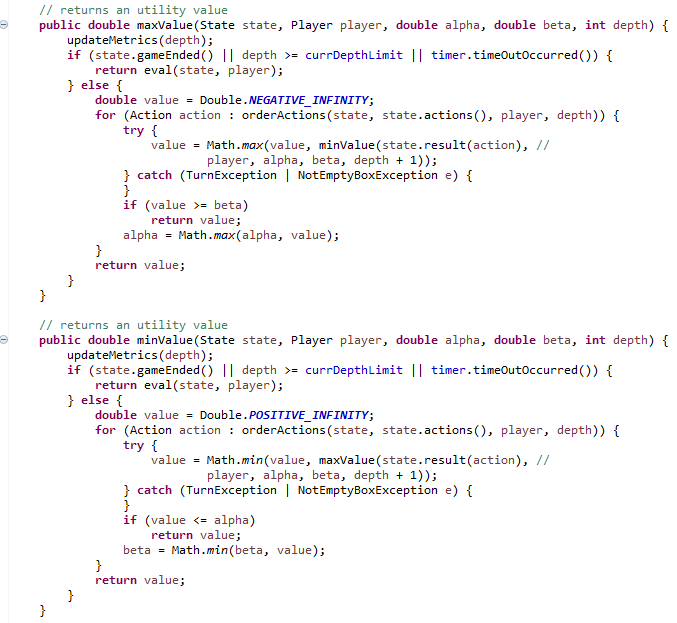
Logicamente, eseguire un’azione genera un **nuovo stato**, serve quindi, sempre nella classe stato un’altra funzione in grado di ritornare lo **stato successivo** a partire dall’esecuzione di un’azione:  


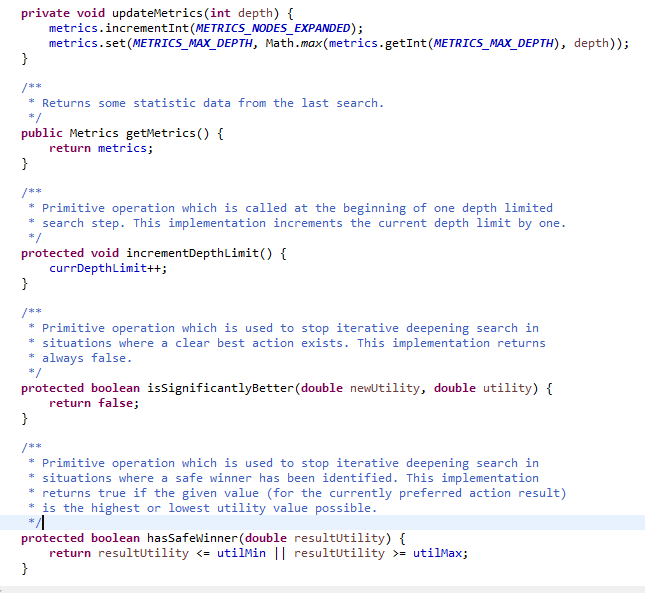
La funzione **result** è IMPORTANTISSIMA perché il suo compito è quello di **far cambiare turno dal giocatore X al giocatore O e viceversa (tramite la setTurn)**.  
Come prima cosa si crea un **nuovo stato copia di quello attuale** con la clone() poi si incrementa il numero di turni e si cambia il turno dal giocatore corrente a quello successivo.  
La riga prima della result semplicemente si occupa di mettere alla riga e alla colonna dell’azione richiesta dal giocatore il **simbolo del giocatore che ha fatto la mossa**. Quindi se la mossa la fa X ci si mette X sennò ci si mette O.

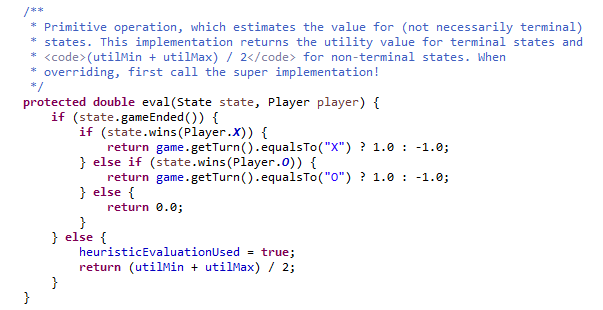
5)Ora che abbiamo la rappresentazione del gioco occorre un **algoritmo di ricerca per la migliore azione possibile da effettuare per ogni giocatore**.  
La tecnica migliore è l’algoritmo MIN-MAX con alfa-beta pruning, in grado di generare un albero che contiene tutte le possibili mosse di entrambi e, sapendo che l’avversario cercherà sempre di fare la mossa migliore, agisce di conseguenza cercando la mossa migliore per lo stato attuale.  
La libreria AIMA è proprio quella che fa al nostro caso, ecco come è strutturata (copy-paste code it’s better)  
NB => Time rappresenta il TEMPO che l’algoritmo impiega per trovare una soluzione in secondi (al massimo). E’ quindi un vincolo temporale!  
  
  


Il costruttore prende in ingresso **uno stato che rappresenta lo stato corrente del game**  
e basa le sue decisioni sulla funzione **makeDecision** che, prendendo in ingresso uno stato restituisce L’AZIONE MIGLIORE a partire da quello stato per (potenzialmente) vincere la partita.







6)Definire una funzione **eval** in grado di assegnare un “PUNTEGGIO” al nodo corrente  


Quando l’algoritmo arriva ad un certo livello è in grado di “stimare” il valore del nodo in base ad un punteggio **attribuito dal programmatore**.  
Nel caso del tris la funzione eval è molto banale, la funzione prende in ingresso lo **stato corrente** (dobbiamo immaginarci di essere ad un certo punto dell’albero in questo istante) e va a **verificare se tale stato è uno stato di vittoria per X o per O.  
NB => La getTurn() restituisce un RIFERIMENTO AL GIOCATORE CHE STA GIOCANDO il turno corrente. (vedi Classe STATE)  
Il punteggio ottenuto dall’esplorazione di UN NODO VINCENTE PER IL GIOCATORE X è di +1 nel caso in vittoria o -1 in caso di sconfitta**.  
Analogamente se stiamo controllando il player O il controllo è lo stesso **:** se il turno corrente è “O” attribuiamo un punteggio +1, sennò -1.  
  
In tutti gli altri casi la mossa è “I don’t actually mind” ed è calcolata con i valori utilMin e utilMax/2 solitamente.  
  
**UtilMin => Data in input al costruttore AIMA rappresenta il valore PEGGIORE per il giocatore che sta giocando il turno  
UtilMax => “” rappresneta il valore MAGGIORE per il giocatore che sta giocando il turno**

Nella classe Aima sono presenti anche le classi Timer, ActionStore e Metrics per l’analisi della ricerca, il tutto è visualizzabile nel codice sorgente trovabile su GITHUB.

7)Costruire un main di prova  
Questo main genera lo stato iniziale (new State()) e fa sì che SOLAMENTE il giocatore “O” sia un giocatore dotato di IA mentre il giocatore “X” come si vede sotto, viene trattato male e fatto giocare in maniera **randomica**. Come prima cosa si stampa lo stato (tutto vuoto)  
Finchè il gioco non termina, se è il turno del giocatore “O” si genera un oggetto AIma in grado di prendere decisioni. Come spiegato nel punto 6 ogni oggetto deve prendere uno stato (quello corrente) e le costanti -infinito e + infinito con il numero di secondi MASSIMI che si vuole dare alla IA per pensare, in questo caso 3 secondi.  
Si sceglie l’azione tramite la funzione makeDecision(statocCorrente) e si **aggiorna lo stato corrente con l’azione eseguita.**  
Nel caso del giocatore randomico si generano semplicemente tutte le azioni possibili con la funzione **actions** e poi si seleziona unazione random da eseguire sul risultato.

