情報論理工学研究室

第5回:
局面・駒石・手の表現

1

ゲームAIの作成

- ゲームAI作成には何が必要か？
  - ルール通りに指せる・打てる
    - 合法手の中で強い手を選べる
  - プレイヤーの手が合法手か判定できる
  - 合法手を指した・打った後の局面を生成できる
  - 終了判定ができる
  - 得点計算・勝敗判定ができる

2

ルール通りに指せる・打てる

- ルール通りに指せる・打てる
  - これができないとそもそもゲームにならない
    - 動かせない場所に駒を動かす
    - 打てない場所に石を打つ
    - 打てない駒・石を打つ
    - 取れない駒・石を取る
    - 手番では無いのに動く
    - 手番なのに動かない

:

でもこれだけでも結構難しい

3

ゲームプログラムの作成

- ルール通りに動くゲームプログラムの作成
  - 必要なクラスを決める
  - 各クラスで必要なメソッドを決める

4

ゲームに必要なクラス

- どんなクラスが必要か？
  - 局面を表現するクラス
  - 駒・石を表現するクラス
  - 入出力を行うクラス
  - 手を表現するクラス
  - 手を指した・打った後の局面を生成するクラス
  - 盤面の評価値を計算するクラス
  - 勝敗判定を行うクラス
  - 様々な定義を行うクラス

:

5

駒・石を表現するクラス

- 駒
  - 駒の種類
  - 誰の駒か
  - 駒の位置
  - 駒の移動範囲

6

コラム:「駒」と「石」

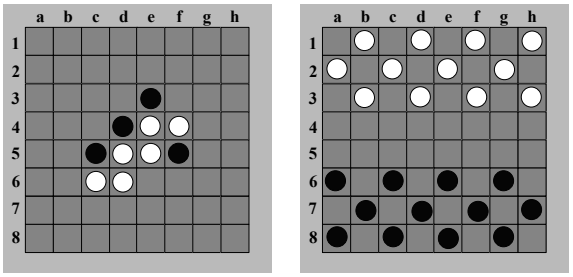
- 駒
  - 盤上を動かす
  - 「指す」
    - 将棋、チェス、チェッカー、バックギャモン

    ※「駒」なのに「打つ」こともある将棋は実は特殊な例
- 石
  - 盤上に置く
  - 「打つ」
    - 囲碁、リバーシ、連珠、三目並べの○と×

7

コラム:「駒」と「石」

リバーシは「石」を使うが…



リバーシの道具でチェッカーをやるならそれは「駒」

8

駒・石の表現

- 石の表現
  - 通常は打った位置から動かない
  - 多くの場合、種類のみで表せる

    ⇒ int型のみで充分な場合が多い
- 駒の表現
  - 駒ごとに動ける範囲が違うことが多い
    - 駒の動ける範囲を表すデータが必要

    ⇒ 駒を表すオブジェクト型が必要な場合がある

9

石の表現:リバーシ

- リバーシの石
  - 黒か白かのみ

int 型で表現
1 : 黒石
-1 : 白石
0 : 空きマス

石を表すクラスは作らずに
局面クラスに直接書き込む

```
class Phase { // 局面を表現するクラス
    int[][] board;
            :
    board = new int [8][8];
            :
}
```

10

駒の表現:将棋

- 将棋の駒
  - 駒ごとに様々な属性を持つ
    - 駒の種類
    - どちらの駒か
    - 成駒か生駒か
    - 盤上の駒か持ち駒か
    - 成れる駒か

```
class Phase {
    Piece[][] board;
            :
    board = new Piece [9][9];
            :
}
```

オブジェクトで表現

11

	Piece		# 駒表現部
	type	: int	# 駒の種類
	canMoves	: int[][]	# 駒の移動可能位置
-	place	: int[][]	# 駒の位置
-	owner	: int	# 駒の持ち主
-	value	: int	# 駒の価値
	Piece()		# コンストラクタ
	toString()	: String	# 駒の文字列表現を返す
	copy()	: Piece	# 駒のコピーを生成
	canPromote()	: boolean	# 成れる駒か
	promote()	: void	# 駒を成る
	promote (type : int)	: void	# 駒を指定した駒になる
	getOwner()	: int	# 駒の持ち主を返す
	isPromote()	: boolean	# 成駒か生駒かを返す
	getValue()	: int	# 駒の価値を返す

12

駒表現の例:将棋

```
Class Piece {
  int type ; // 駒の種類
 /*
   駒の種類、
   先手駒か後手駒のどちらであるか、
   生成か成駒のどちらであるか、等を表す
  */
  static int[][] canMoves; // 各駒が動ける方向
 /*
   駒の種類ごとに、その駒が動ける方向を表す
   その方向へ1マスのみ動けるのか、いくらでも動けるのかも区別する
 */
}
```

13

駒表現の例：将棋

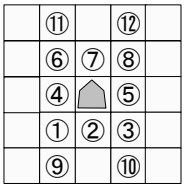
駒の種類を表す表を作成する

00	01	02	03	04	05	06	07
	歩	香	桂	銀	金	角	飛
08	09	0A	0B	0C	0D	0E	0F
玉	と	杏	圭	全		馬	龍
10	11	12	13	14	15	16	17
	歩	香	桂	銀	金	角	飛
18	19	1A	1B	1C	1D	1E	1F
玉	と	杏	圭	全		馬	龍

01〜0F : 先手駒
11〜1F : 後手駒
*1〜*8 : 生駒
*9〜*F : 成駒

14

駒表現の例：将棋

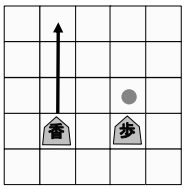


	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	⑪	⑫
歩	0	0	0	0	0	0	1	0	0	0	0	0
香	0	0	0	0	0	0	2	0	0	0	0	0
桂	0	0	0	0	0	0	0	0	0	0	1	1
銀	1	0	1	0	0	1	1	1	0	0	0	0
金	0	1	0	1	1	1	1	1	0	0	0	0
角	2	0	2	0	0	2	0	2	0	0	0	0
飛	0	2	0	2	2	0	2	0	0	0	0	0
玉	1	1	1	1	1	1	1	1	0	0	0	0

各駒の動ける方向を定義する
0:その方向へは動けない
1:その方向へ1マス動ける
2:その方向へいくらでも動ける

15

駒表現の例：将棋



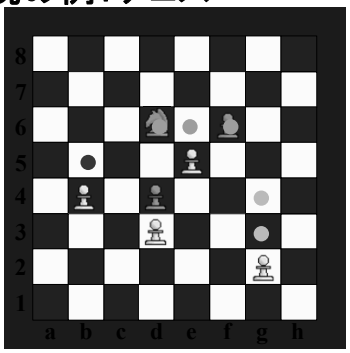

```
if (移動可能[前] == 1) {
  if (盤[x][y-1] == 空)
    (x,y-1) へ移動可
  else if (盤[x][y-1] == 敵駒)
    (x,y-1)へ駒を取って移動可
} else if (移動可能[前] == 2) {
  for (v=y-1; 盤[x][v]==空; --v)) {
    (x,v)へ移動可
  }
  if (盤[x][v]==敵駒) {
    (x,v)へ駒を取って移動可
}
```

16

駒表現の例:チェス

ポーンの移動

1. 前方のマスが空いていれば1マス進める
2. 斜め前に敵駒があればその駒と取って進める
3. 初期位置から移動していないポーンは2マス進める



場合分けが必要

17

駒表現の例：チェス

0:その方向へは動けない
1:その方向へ1マス動ける
2:その方向へいくらでも動ける
3: 敵駒が無い場合のみ1マス動ける
　さらに初期位置にいる場合のみ2マス動ける
4: 敵駒がある場合のみ1マス動ける

	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	⑪	⑫	⑬	⑭	⑮	⑯
P	0	0	0	0	0	4	3	4	0	0	0	0	0	0	0	0
R	0	2	0	2	2	0	2	0	0	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
B	2	0	2	0	0	2	0	2	0	0	0	0	0	0	0	0
Q	2	2	2	2	2	2	2	2	0	0	0	0	0	0	0	0
K	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

18

駒の文字列表現を返すメソッド

- toString()メソッド
  - 駒の文字列表現を返す

```
String toString() {
  switch (type) {
    case 01 : return " 歩";
    case 02 : return " 香";
    case 03 : return " 桂";
              :
```

```
              :
    case 11 : return "v歩"
    case 12 : return "v香"
              :
    default : return "   ";
  }
```

19

駒が成れるかを返すメソッド

- canPromote() メソッド
  - 駒が成れるかを返す

```
boolean canPromote() {
  switch (type) {
    case 歩 : case 香 : case 桂 : case 銀 : case 角 : case 飛 :
      return true;
    default :
      return false;
  }
}
```

20

駒を成るメソッド

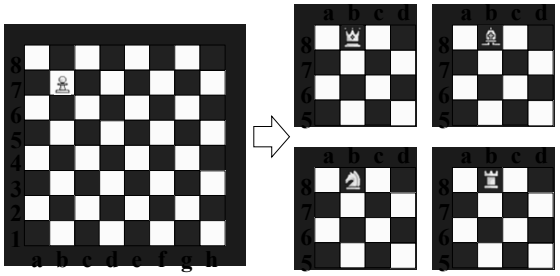
- promote() メソッド
  - 駒を成る

```
void promote() {
  if (!canPromote()) error(); // 成れない駒はエラー
  switch (type) {
    case 歩 : type = と; break;
    case 香 : type = 杏; break;
    case 桂 : type = 圭; break;
              :
  }
}
```

21

チェスの昇格

- 昇格
  - 最前列に到達したポーンはキング以外の任意の駒に成れる



成る駒の種類を指定する必要がある

22

駒を成るメソッド

- promote() メソッド
  - 駒を指定した種類の駒になる

```
void promote (int newType) {
  if (type != PAWN) error(); // 成れるのはポーンだけ
  type = newType;
}
```

23

局面を表現するクラス

- 局面
  - 変数
    - 盤上にある駒・石の種類と位置
    - 持ち駒
    - 先手・後手
    - 同一局面になった回数
  - メソッド
    - 表示
    - コピー
    - 駒・石の初期配置
    - 同一局面か？

24

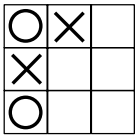
Phase		# 局面表現部
board	: int[][]	# 盤面
turn	: int	# 手番
- value	: int	# 局面の評価値
- captured	: int[]	# 持ち駒
- lastMove	: Move	# 直前の手
Phase ()		# コンストラクタ
show()	: void	# 盤面表示
copy()	: Phase	# 局面のコピーを生成
set (piece : Piece, place : int[][])	: void	# 指定した駒を配置
initiallySet()	: void	# 駒を初期配置
equals (phase : Phase)	: Boolean	# 局面の同一判定
nextPhase (move : Move)	: Phase	# 1手後の局面を生成
isWin()	: int	# 勝敗判定
getValue()	: int	# 局面の評価値

25

盤面の表現

- 盤面の表現
  - 盤面は2次元配列で表現できる

int board[][] = new int[3][3];



1	-1	0
-1	0	0
1	0	0

○:1
×:-1
空:0

26

盤面の表現

使用する駒・石が単純なものなら
int 型で表現するのが簡単

初期値無し (三目並べ)

```
int[][] board = new int [3][3];
```

三目並べの場合
空=0, ○=1, ×=-1

初期値有り (将棋)

```
int[][] board = {{12, 13, 14,...}
                 {00, 17, 00,...}
                 {11, 11, 11,...}};
```

将棋の場合
空=00,
歩=01, 香=02, 桂=03, …
=11, 王=12, 玉=13, …

27

盤面の表現

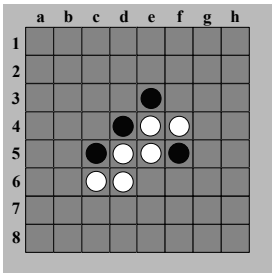
複雑な駒・石を使用する場合は
駒を表すオブジェクト型の配列にする

```
駒[][] 盤 = new 駒 [9][9];
```

```
駒[][] 盤 = {{new 駒(香), new 駒(桂), new 駒(銀), …
             {null,       new 駒(飛), null,       …
             {new 駒(歩), new 駒(歩), new 駒(歩), …
                  :
```

28

盤面表現の例:リバーシ



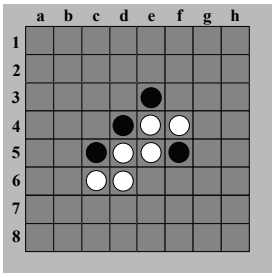
黒番

```
board [][] = {
 {0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 1, 0, 0, 0},
 {0, 0, 0, 1,-1,-1, 0, 0},
 {0, 0, 1,-1,-1, 1, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0},
};
turn = 1;
```

29

盤面表現の例:リバーシ

多くのゲームでは盤面を一回り大きくして
周囲を「壁」にしておくと便利

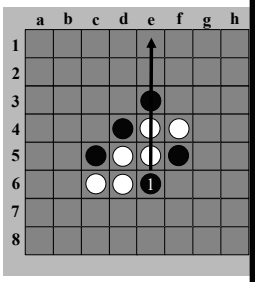



```
board [][] = {
 {∞,∞,∞,∞,∞,∞,∞,∞,∞,∞},
 {∞,0, 0, 0, 0, 0, 0, 0, 0,∞},
 {∞,0, 0, 0, 0, 0, 0, 0, 0,∞},
 {∞,0, 0, 0, 0, 1, 0, 0, 0,∞},
 {∞,0, 0, 0, 1,-1,-1, 0, 0,∞},
 {∞,0, 0, 1,-1,-1, 1, 0, 0,∞},
 {∞,0, 0, 0, 0, 0, 0, 0, 0,∞},
 {∞,∞,∞,∞,∞,∞,∞,∞,∞,∞},
};
```

30

盤面表現の例:リバーシ

「壁」を用いない場合

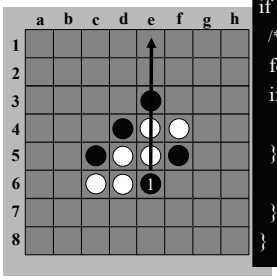



```
if (盤[x][y-1]==白) {
 /* 上方向に白石が続く限り探索 */
 for (v=y-1; v≧0; --v)) {
   if (盤[x][y] ≠ 白) break;
 }
 if (v<0) { /* 盤外の場合 */
  上方向の石はひっくり返せない
 } else if (盤[x][v]==黒) {
   間の石をひっくり返す
 } else { /* 空マスの場合 */
   上方向の石はひっくり返せない
 }
}
```

31

盤面表現の例:リバーシ

「壁」を用いる場合



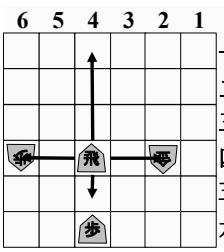
盤外に出たかの判定が不要

```
if (盤[x][y-1]==白) {
 /* 上方向に白石が続く限り探索 */
 for (v=y-1; 盤[x][v]==白; --v));
 if (盤[x][v]==黒) {
   間の石をひっくり返す
 } else { /* 空マスまた壁の場合 */
   上方向の石はひっくり返せない
 }
}
```

32

盤面表現の例:将棋

「壁」を用いない場合

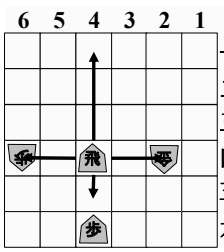



```
/* 上方向に空マスが続く限り探索 */
for (v=y-1; v≧0; --v)) {
 if (盤[x][y] == 空マス)
 (x,v)へ移動する手を合法手に加える
 else break; /* 自駒か敵駒がある場合 */
}
if (v<0) {
 壁に到達した
} else if (盤[x][v]==敵駒) {
 (x,v)の駒を取る手を合法手に加える
} else { /* 自駒の場合 */
 (x,v)へは移動できない
}
```

33

盤面表現の例:将棋

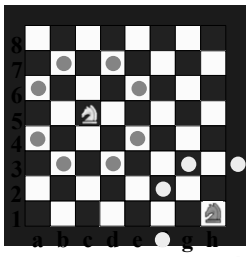
「壁」を用いる場合




```
/* 上方向に空マスが続く限り探索 */
for (v=y-1; 盤[x][v]==空; --v)) {
 (x,v)へ移動する手を合法手に加える
}
if (盤[x][v]==敵駒) {
 (x,v)の駒を取る手を合法手に加える
} else { /* 自駒または壁の場合 */
 (x,v)へは移動できない
}
```

34

盤面表現の例:チェス



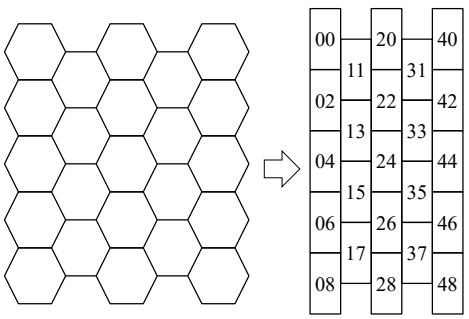
ナイト♘は離れたマスに飛べる

```
{{∞,∞,∞,∞,∞,∞,∞,∞,∞,∞,∞,∞},
 {∞,∞,∞,∞,∞,∞,∞,∞,∞,∞,∞,∞},
 {∞,∞, 0, 0, 0, 0, 0, 0, 0, 0,∞,∞},
 {∞,∞, 0, 0, 0, 0, 0, 0, 0, 0,∞,∞},
 {∞,∞, 0, 0, 0, 0, 0, 0, 0, 0,∞,∞},
 {∞,∞, 0, 0, 3, 0, 0, 0, 0, 0,∞,∞},
 {∞,∞, 0, 0, 0, 0, 0, 0, 0, 0,∞,∞},
 {∞,∞, 0, 0, 0, 0, 0, 0, 0, 0,∞,∞},
 {∞,∞, 0, 0, 0, 0, 0, 0, 0, 0,∞,∞},
 {∞,∞, 0, 0, 0, 0, 0, 0, 0,-3,∞,∞},
 {∞,∞,∞,∞,∞,∞,∞,∞,∞,∞,∞,∞},
 {∞,∞,∞,∞,∞,∞,∞,∞,∞,∞,∞,∞}}
```

壁を二重にしておく

35

ヘクスマップの場合の盤面表現

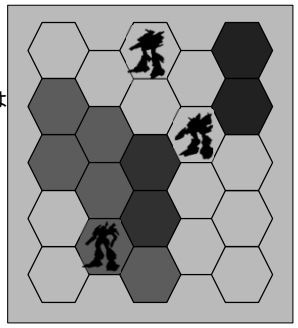


六角形はサイズ1*2の長方形で表現できる

36

盤面の表現の例：バトルテック

バトルテック
- 巨大ロボットの戦闘ゲーム
- 各ロボットの武装・装甲等はゲーム中に変化



37

盤面の表現の例：バトルテック

機種：フェニックスホーク
重量：45t

移動

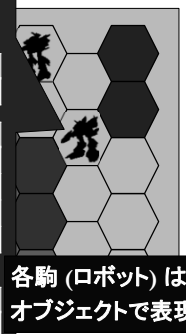
歩行	走行	ジャンプ
6	9	6

武器

武器	威力	射程	弾薬	位置	破損
大型レーザー	8	15	-	右腕	
中型レーザー	5	9	-	右腕	×
中型レーザー	5	9	-	左腕	
マシンガン	2	3	192	左腕	

装甲

頭	正面	背面	右腕	左腕	右脚	左脚
6	23	5	0	10	15	9

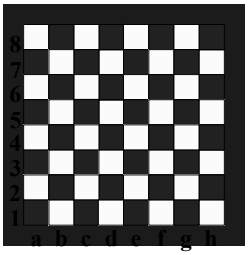


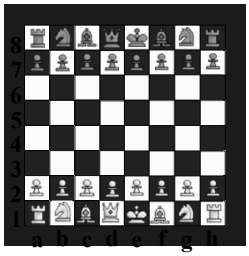
各駒 (ロボット) はオブジェクトで表現

38

局面クラスのコンストラクタ

- コンストラクタは2種類作っておくと便利
  - 空マスのみの盤面を生成
  - 初期局面の盤面を生成





39

コンストラクタの例

```
public class Phase () {
  int[][] board:        // 盤面
  int turn;             // 手番
  int value;            // 評価値
  Move lastMove:   // 直前の手

  Phase() {
    board = new int [SIZE][SIZE];
    for (int[] n : board) for (int m : n) m = EMPTY; // 盤を空白で埋める
    turn = WHITE;     // 先手は白番
    value = 0;            // 初期状態では評価値は0
    lastMove = null;   // 直前の手は無し
  }
}
```

40

コンストラクタの例

```
Phase (int setType) {
  switch (setType) {
  case 0 :  // 引数が0なら空白の盤を作成
    board = new int [SIZE][SIZE];
    for (int[] n : board) for (int m : n) m = 0; // 盤を空白で埋める
    break;
  case 1 : // 引数が1なら初期配置の盤を作成
    board = {{R, N, B, Q, K, B, N, R},
             {P, P, P, P, P, P, P, P},
             {0, 0, 0, 0, 0, 0,, 0, 0},
                          :
    break;
                          :
```

41

1次元配列での表現

盤面は1次元配列で表現してもいい

int a[X][Y]

1	-1	0
-1	0	0
1	0	0

int b[X*Y]

1	-1	0
-1	0	0
1	0	0

座標 (i, j) は i+jX で表現する　　(1, 2) = 7

方向 (u,v) も u+vX で表現する　　(-1, +1) = 2

42

1次元配列での表現

- 1次元配列を使う利点
  - 2次元配列よりも処理が速い
  - 座標を数値1つで表現できる
  - 方向も数値1つで表現できる
  - clone()メソッドでコピーできる
- 1次元配列を使う注意点
  - 端の処理に注意が必要
  - 座標・方向の対応に注意が必要
  - 1次元でもオブジェクト型の配列はclone()では無理

43

1次元配列での表現の例:3目並べ

int a[10];

7	8	9
4	5	6
1	2	3

(a[0]は使用しない)

```
int place;
while (true) { // 適切な位置が選択されるまでループ
  String inputString = keyBoardScanner.next();
 try {
   place = Integer.parseInt (inputString);
 } catch (NumberFormatException e) {
    continue; // 整数以外
 }
 if (place <1 || 9<place)  {
   continue; // 範囲外
 }
 break;
}
```

座標の入力が1回ですむ

44

局面の表示

- show()メソッド
  - 盤面、持ち駒、手番等を表示する

```
void show() {
  for (int i=0; i<SIZE; ++i) {
    for (int j=0; j<SIZE; ++j)
       System.out.print (board[i][j]);
    System.out.println();
  }
  System.out.println (turn);
                :
}
```

45

局面のコピー

- copy()メソッド
  - 盤面、手番、持ち駒等を全てコピーする
    (注意)盤面に2次元配列を用いている場合は要素ごとにコピーする必要がある

46

局面のコピー

```
Phase copy() {
  Phase newPhase = new Phase();
  for (int i=0; i<SIZE; ++i)
    for (int j=0; j<SIZE; ++j);
      newPhase.board[i][j] = this.board[i][j];
  newPhsse.turn = this.turn;
  newPhase.value = this.value;
                :
  return newPhase;
}
```

要素ごとにコピー

47

局面のコピー

盤面が1次元配列で表現されている場合

```
Phase copy() {
  Phase newPhase = new Phase();
  newPhase.board = this.board.clone();
  newPhsse.turn = this.turn;
  newPhase.value = this.value;
                :
  return newPhase;
}
```

1次元配列ならclone()メソッドでいい

48

駒・石の配置

- set()メソッド
  - 指定した駒・石を指定の位置にセットする

```
void set (int type, int x, int y) {
  board [x][y] = type;
}
```

```
void set (int type, int x, int y) {
  board [x][y] = new Piece (type);
}
```

49

駒・石の削除

- remove()メソッド
  - 指定した位置の駒・石を削除する

```
void remove (int x, int y) {
  board [x][y] = EMPTY;
}
```

将棋では、取った駒を
持ち駒に加える処理も必要

50

駒・石の移動

- move()メソッド
  - 指定した位置の駒・石を指定した位置に移動する

```
void move (int x, int y, int u, int v) {
  board [u][v] = board [x][y];
  board [x][y] = EMPTY;
}
```

51

駒・石の全削除

- clean()メソッド
  - 全ての駒・石を削除する

```
void clean () {
  for (int i=0; i<SIZE; ++i)
    for (int j=0; j<SIZE; ++j)
      board [i][j] = EMPTY;
}
```

52

駒・石の初期配置

- initiallySet()メソッド
  - 駒・石を初期位置にセットする

```
void initiallySet () {
  clean();
  board [1][1] = ROOK;
  board [2][1] = KNIGHT;
  board [3][1] = BISHOP;
          :
}
```

53

局面の同一判定

- equals()メソッド
  - 同一の局面か判定する

```
boolean equals (Phase phase) {
  for (int i=0; i<SIZE; ++i)
    for (int j=0; j<SIZE; ++j)
      if (this.board[i][j] ≠ phase.board[i][j])
        return false; // 1箇所でも異なればfalse
  if (this.turn ≠phase.turn) return false;
                :
  return true; // 全て同じならtrue
}
```

54

1手後の局面を生成

- nextPhase()メソッド
  - 指定した手を指した後の局面を生成する

```
Phase nextPhase (Moves nextMoves) {
  Phase nextPhase = this.copy();  // 現在の局面をコピー
  nextPhase.move (nextMoves); // 指定した手を指す
  nextPhase.turn = !this.turn;   // 手番を交代する
                  :
  return nextPhase;
}
```

55

勝敗判定

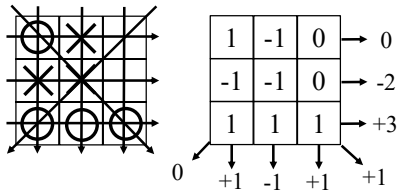
- isWin()メソッド
  - 勝敗判定する

```
int isWin () {
  勝敗判定を行う
  先手勝ちなら1, 後手勝ちなら-1,
  まだ勝負がついていないなら0を返す
}
```

56

3目並べの勝利判定

- 盤面の勝利判定
  - 縦横斜めの各列で○×が3つ並んだか調べる



各列の和を求める

+3:○の勝ち -3:×の勝ち

57

局面の評価値

- getValue()メソッド
  - 局面の評価値を返す

```
int getValue () {
  現在の局面からどちらが優勢かを返す
  先手優勢なら正、後手優勢なら負の値
  先手勝ちなら+∞、後手勝ちなら-∞
}
```

58

局面の評価値

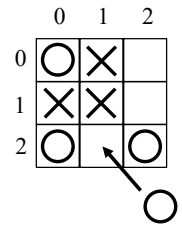
- getValue()メソッド
  - 指定した手数を先読みした上で局面の評価値を返す

```
int getValue (int depth) {
  現在の局面からdepth先まで読んでどちらが優勢かを返す
  先手優勢なら正、後手優勢なら負の値
  先手勝ちなら+∞、後手勝ちなら-∞
}
```

59

次の手を表現するクラス

- 次の手
  - 駒・石の種類
  - 動かす・置く位置




```
class Puts {
  int[] place;
  int type:
}
```

place = {1, 2}

type = NOUGHT

60

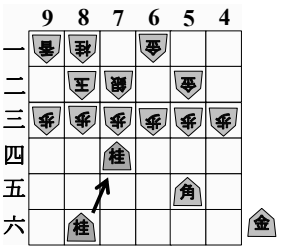
次の手を表すクラス

```
class Puts {
  int[] place; // 石を置く位置
  int type: // 置く石の種類

  Puts (int[] place, int type) {
    this.place = place.clone();
    this.type = type;
}
```

61

次の手を表すクラス



8六の桂を7四へ移動

```
class Moves {
  int[] beforePlace;
  int[] afterPlace;
  int type:
}
```

beforePlace = {8, 6}
afterPlace = {7, 4}
type = KEIMA

62

次の手を表すクラス

```
class Moves {
  int[] beforePlace; // 駒の元の位置
  int[] afterPlace; // 駒を動かす位置
  int type: // 動かす駒の種類

  Moves (int[] beforePlace, int[] afterPlace) {
    this.beforePlace = beforePlace.clone();
    this.afterPlace = afterPlace.clone();
    this.type = board [beforePlace];
}
```

63

手の同一判定

- equals() メソッド
  - 同一の手か判定

```
boolean equals (Moves moves) {
  if (this.beforePlace ≠ moves.beforePlace)
    return false; // 1箇所でも異なればfalse
  if (this.afterPlace ≠ moves.afterPlace)
    return false;
  if (this.type ≠ moves.type)
    return false;
                :
  return true; // 全て同じならtrue
}
```

64

合法手の判定

- isLegalMoves()メソッド
  - 合法手かどうか判定する

```
boolean isLegalMoves () {
  ルール上認められる手かを返す
  動かせない駒を動かす、動かせない位置に動かす、
  王手を放置している、等の場合はfalseを返す
}
```

65

合法手を生成するクラス

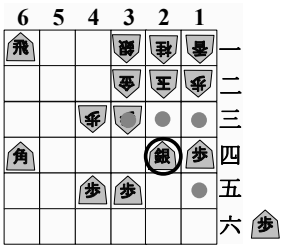
- 与えられた局面で可能な合法手を生成する
  - 合法手リストを返す
  - 合法手リストに指定した手を加える
  - 合法手リストから指定した手を取り除く

66

	GenerateMoves	# 合法手生成部
- phase	: Phase	# 局面
- MovesList	: ArrayList	# 合法手のリスト
GenerateMoves (phase : Phase)		# コンストラクタ
- addMoves (moves : Moves)	: void	# リストに手を加える
- removeMoves (moves : Moves)	: void	# リストから手を取り除く
- removeSelfMate ()	: void	# リストから自殺手を取り除く
- generateMoves (place : int[])	: void	# 指定の位置にある駒を動かす手をリストに加える
- generatePuts (type : int)	: void	# 指定した種類の石を打つ手をリストに加える

67

指定した駒を動かす手を生成



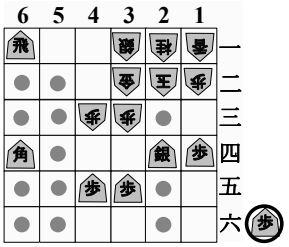
gererateMoves (2, 4);

2四の銀を動かす手を
リストに加える

▲３三銀成　: Moves (2,4,3,3, t)
▲３三銀不成 : Moves (2,4,3,3, f)
▲２三銀成　: Moves (2,4,2,3, t)
▲２三銀不成 : Moves (2,4,2,3, f)
▲１三銀成　: Moves (2,4,1,3, t)
▲１三銀不成 : Moves (2,4,1,3, f)
▲１五銀　　: Moves (2,4,1,5)

68

指定した駒を打つ手を生成



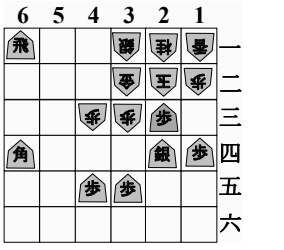
gereratePuts (FU);

歩を打つ手を
リストに加える

▲６二歩　: Puts (FU, 6, 2)
▲５二歩　: Puts (FU, 5, 2)
▲６三歩　: Puts (FU, 6, 3)
▲５三歩　: Puts (FU, 5, 3)
▲２三歩　: Puts (FU, 2, 3)
▲５四歩　: Puts (FU, 5, 4)
▲６五歩　: Puts (FU, 6, 5)
:

69

自殺手を削除



▲２三歩まで

removeSelfMate ();

△２三同金以外の手を
リストから取り除く

70

合法手を生成

```
class GenerateMoves {
  ArrayList<Moves> movesList;

  GenerateMoves (Phase phase) {
    movesList = new ArrayList<Moves>;
    for (int i=0; i<SIZE; ++i) {     // 盤上の各駒を動かす手をリストに加える
      for (int j=0; j<SIZE; ++j) {
        movesList.add (generateMoves (i, j));
    for (int type : acaptredList) {  // 持ち駒を打つ手をリストに加える
      movesList.add (generatePuts (type);
    removeSelfMate();                // 自殺手を取り除く
}
```

71

合法手生成:3目並べ

int a[10];

7	8	9
4	5	6
1	2	3

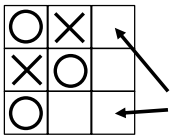
(a[0]は使用しない)

```
int place;
while (true) { // 適切な位置が選択されるまでループ
  String inputString = keyBoardScanner.next();
 try {
  place = Integer.parseInt (inputString);
 } catch (NumberFormatException e) {
   continue; // 整数以外
 }
 if (place <1 ‖ 9<place)  continue; // 範囲外
 if (a[place] ≠0) )  continue; // 空マスではない
 break;
}
```

72

宿題:3目並べのリーチ判定

- 3目並べでリーチ(あと1箇所置けば勝てる)の判定方法を考えよ

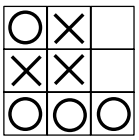


○はここに置ければ勝ち
＝○のリーチ

73

3目並べの勝利判定

- 盤面の勝利判定
  - 縦横斜めの各列で○×が3つ並んだか調べる



1	-1	0	0
-1	-1	0	-2
1	1	1	+3
0	+1	-1	+1

各列の和を求める

+3:○の勝ち -3:×の勝ち

リーチのときは各列の和はいくらになる？

74