# 情報論理工学研究室

第6回:
リバーシの合法手生成

1

## 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

2

## 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

:

3

## 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

4

## 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型のみで十分な場合が多い
- 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

5

## 石の表現:リバーシ

- リバーシの石
  - 白か黒かのみ

int 型で表現
1 : 黒石
-1 : 白石
0 : 空きマス
∞ : 壁

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 [10][10];
            :
}
```

6

## コラム：石の表現

- 何故黒=1,白=-1にする？
  - 石をひっくり返す処理が以下の命令でできる

```
board[x][y] = -board[x][y];
```

  - 符号を反転すればいい
    ＝直観的にひっくり返すイメージ通り
- 何故壁＝∞にする？
  - 処理の都合上設けた値
    (ゲームに必要な値である黒、白、空マスとは異なる)
    ⇒明らかに“異常な”値にしておいた方が分かり易い

7

## 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？

8

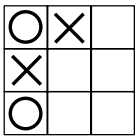
| Phase | | # 局面表現部 |
|---|---|---|
| board | : int[][] | # 盤面 |
| turn | : int | # 手番 |
| - value | : int | # 局面の評価値 |
| - lastMove | : Move | # 直前の手 |
| Phase () | | # コンストラクタ |
| show() | : void | # 盤面表示 |
| copy() | : Phase | # 局面のコピーを生成 |
| set (disc : int, position: int[][]) | : void | # 指定した石を配置 |
| initiallySet() | : void | # 石を初期配置 |
| 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 |

9

## 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];
```



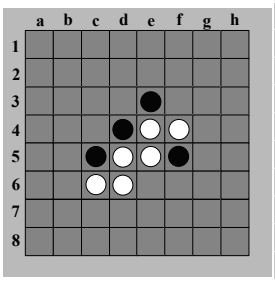
| | | |
|---|---|---|
| 1 | -1 | 0 |
| -1 | 0 | 0 |
| 1 | 0 | 0 |

○:1
×:-1
空:0

10

## 盤面の表現

盤面をサイズ10×10の2次元配列 int[10][10] で表現
周囲には「壁」を置く




| y \ x | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 1 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ |
| 2 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ |
| 3 | ∞ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | ∞ |
| 4 | ∞ | 0 | 0 | 0 | 1 | -1 | -1 | 0 | 0 | ∞ |
| 5 | ∞ | 0 | 0 | 1 | -1 | -1 | 1 | 0 | 0 | ∞ |
| 6 | ∞ | 0 | 0 | -1 | -1 | 0 | 0 | 0 | 0 | ∞ |
| 7 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ |
| 8 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ |
| 9 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |

11

## Phase クラス

```
/* 局面を定義するクラス */
class Phase {
  public static final int BLACK = 1; // 黒石
  public static final int WHITE = -1; // 白石
  public static final int EMPTY = 0;  // 空マス
  public static final int WALL = Integer.MAX_VALUE; //壁

  public int[][] board; // 盤面
  public int turn;      // 手番
                        :
```

12

## Phase クラス

```
/* コンストラクタ */
public Phase () {
    board = new int[10][10]; // 盤面を2次元配列で表現
    /* 盤面の周囲を壁で、内側を空マスで埋める */
    for (int x=0; x<10; ++x) board[x][0] = WALL:
    for (int y=1; y<9; ++y) {
        board[0][y] = WALL;
        for (int x=1; x<9; ++x) board[x][y] = EMPTY;
        board[9][y] =WALL;
    }
    for (int x=0; x<10; ++x) board[x][9] = WALL;
    turn = BLACK; // 先手は黒盤
}
```

13

## Point クラス

```
/* 座標を定義するクラス */
class Point {
  public int x, y; // 座標

  /* 引数無しのコンストラクタ */
  public Point () {
     this.x = 0; this.y = 0;
   }
  /* 引数有りのコンストラクタ */
  public Point (int x, int y) {
     this.x = x;  this.y = y;
   }
}
```

14

## 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

15

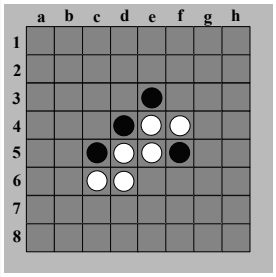
## 1次元配列での表現

- 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 (Point クラスが不要)
  - 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

16

## 盤面の表現(1次元配列)

盤面はサイズ100の1次元配列 int[100] で表現




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 10 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ |
| 20 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ |
| 30 | ∞ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | ∞ |
| 40 | ∞ | 0 | 0 | 0 | 1 | -1 | -1 | 0 | 0 | ∞ |
| 50 | ∞ | 0 | 0 | 1 | -1 | -1 | 1 | 0 | 0 | ∞ |
| 60 | ∞ | 0 | 0 | -1 | -1 | 0 | 0 | 0 | 0 | ∞ |
| 70 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ |
| 80 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ |
| 90 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |

こちらの方が多分速い

17

## Phase クラス(1次元配列の場合)

```
/* コンストラクタ */
public Phase () {
    board = new int[100]; // 盤面を1次元配列で表現
    /* 盤面の周囲を壁で、内側を空マスで埋める */
    for (int x=0; x<10; ++x) board[x] = WALL:
    for (int y=10; y<90; y+=10) {
        board[y] = WALL;
        for (int x=1; x<9; ++x) board[x+y] = EMPTY;
        board[9+y] =WALL;
    }
    for (int x=0; x<10; ++x) board[x+90] = WALL;
    turn = BLACK; // 先手は黒盤
}
```

18

# 盤面の表現(1次元配列)

サイズ160の1次元配列 int [160] の方が高速かも…

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 1 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 2 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 3 | ∞ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 4 | ∞ | 0 | 0 | 0 | 1 | -1 | -1 | 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 5 | ∞ | 0 | 0 | 1 | -1 | -1 | 1 | 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 6 | ∞ | 0 | 0 | -1 | -1 | 0 | 0 | 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 7 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 8 | ∞ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 9 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |

19

# サイズ$2^n$の一次元配列での表現

- サイズ$2^n$を使う利点
  - ビット計算が利用可能(かもしれない)

| 方向ベクトル | | |
|---|---|---|
| -17 | -16 | -15 |
| -1 | 0 | +1 |
| +15 | +16 | +17 |

上下方向の計算が±16の加減算
±10より速い(かもしれない)

- サイズ$2^n$を使う欠点
  - メモリが余分に必要
    - でもサイズ100も160もメモリ使用量は同じかも…
      (どちらも256割り当てになるかも)

処理系に依存

20

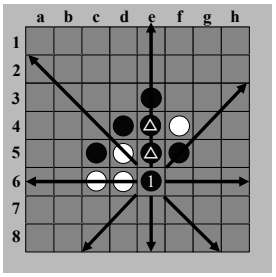
# 盤面表示メソッド

```
/* 盤面表示 */
public void showBoard () {
    for (int y=0; y<10; ++y)
        for (int x=0; x<10; ++x) {
            switch (board[x][y] {
                case BLACK : System.out.print (“●”); break;
                case WHITE : System.out.print (“○”); break;
                case EMPTY : System.out.print (“ ”); break;
                case WALL : System.out.print (“■”); break;
            }
            System.out.println();
        }
}
```

21

# 合法手の生成

置いた石から8方向にひっくり返せるかチェック



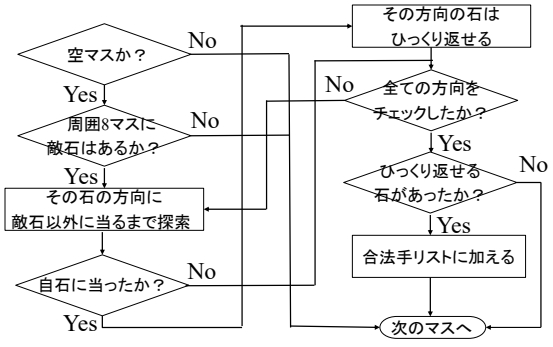

隣に敵石があり、
かつその方向に
自石があれば
ひっくり返せる

ひっくり返せる
石があるならば
そのマスに置ける

22

# 合法手の生成

各マスに対して以下の処理を行う




23

# 合法手の判定

- isLegalMoves()メソッド
  - 合法手かどうか判定する

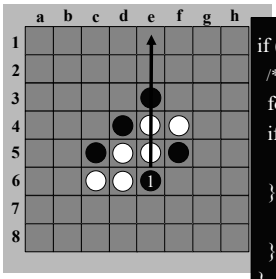
```
boolean isLegalMoves () {
  ルール上認められる手かを返す
  動かせない駒を動かす、動かせない位置に動かす、
  王手を放置している、等の場合はfalseを返す
}
```

24

## 合法手の判定

上方向への探索

(x, y) に黒石を置いた場合




```
if (盤[x][y-1]==白) {
  /* 上方向に白石が続く限り探索 */
  for (v=-2; 盤[x][y+v]==白; --v));
  if (盤[x][y+v]==黒) {
    間の石をひっくり返せる
  } else { /* 空マスまた壁の場合 */
    上方向の石はひっくり返せない
  }
}
```

25

## 8方向の表現

8つの方向ベクトルを表す配列を用意

| | | |
|---|---|---|
| (-1,-1) | (0, -1) | (+1,-1) |
| (-1,0) | | (1, 0) |
| (-1, +1) | (0, +1) | (+1, +1) |

| | 左上 | 上 | 右上 | 左 | 右 | 左下 | 下 | 右下 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| vx[] | -1 | 0 | +1 | -1 | +1 | -1 | 0 | +1 |
| vy[] | -1 | -1 | -1 | 0 | 0 | +1 | +1 | +1 |

```
static final int[] vx = {-1, 0, 1, -1, 1, -1, 0, 1};
static final int[] vy = {-1, -1, -1, 0, 0, 1, ,1, 1};
```

マス (x,y) の周囲8マスは (x+vx[i], y+vy[i]) で表される

26

## 合法手の判定

```
/* マス(x,y)に石を置けるか判定するメソッド */
boolean isLegalMove (Point point, int color) {
  int x = point.x, y=point.y;
  if (borad [x][y] != EMPTY) return false;
  for (int i=0; i<8; ++i)
    if (board[x+vx[i]][y+vy[i]] == -color) { // 隣の石が敵石の場合
      int k=2;
      // 敵石以外に当るまで探索
      while (board[x+k*vx[i]][y+k*vy[i]] == -color) ++k;
      if (board[x+k*vx[i]][x+k*vy[i]] == color) // 自石に当った場合
        return true;
    }
  return false:
}
```

27

## 8方向の表現(1次元配列の場合)

8つの方向ベクトル(1次元配列の場合)

| | | |
|---|---|---|
| -11 | -10 | -9 |
| -1 | | +1 |
| +9 | +10 | +11 |

| | 左上 | 上 | 右上 | 左 | 右 | 左下 | 下 | 右下 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| v[] | -11 | -10 | -9 | -1 | +1 | +9 | +10 | +11 |

```
static final int[] v = {-11, -10, -9, -1, 1, 9, 10, 11};
```

マス (p) の周囲8マスは (p+v[i]) で表される

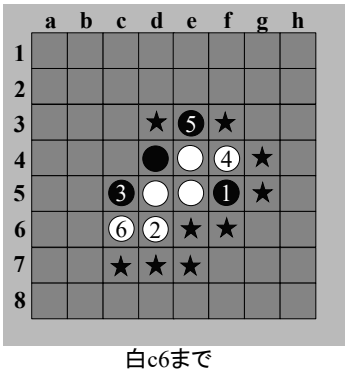
28

## 合法手の判定(1次元配列の場合)

```
/* マス(x,y)に石を置けるか判定するメソッド */
boolean isLegalMove (int p, int color) {
  if (borad [p] != EMPTY) return false;
  for (int i=0; i<8; ++i)
    if (board[p+v[i]] == -color) {
      int k=2;
      // 敵石以外に当るまで探索
      while (board[p+k*v[i]] == -color) ++k;
      if (board[p+k*v[i]] == color)
        return true; // 自石ならtrue
    }
  return false:
}
```

29

## 合法手リストの生成




黒石を置けるのは★の9か所

合法手

(d,3) (f,3) (g,4)
(g,5) (e,6) (f,6)
(c,7) (d,7) (e,7)

白c6まで

30

## 合法手リストの生成

```
/* 合法手リストを生成する */
ArrayList<Point> GenerateMoveList (int color) {
   ArrayList<Point> moveList = new ArrayList<Point>;
  for (int y=1; y<9; ++y) {
     for (int x=1; x<9; ++x) {
        Point point = new Point (x, y);
        boolean canMove = isLegalMove (point, color);
        if (canMove) moveList.add (point); // リストに加える
     }
  }
  return moveList;
}
```

31

## パスの判定

- パスの判定
  - 打てる手が無い場合はパス
  - 双方パスするとゲーム終了

```
                                    :
ArrayList<Point> moveList = generateMoveList (color);
 if (moveList.isEmpty()) { // 合法手が無い場合
   moveList = generateMoveList (-color); // 相手の合法手をチェック
   if (moveList.isEmpty()) { // 相手も合法手が無い
     gemeSet(); // ゲーム終了
   } else pass(); // パス
 }
                                    :
```

32

## 手の入力

```
/* キーボードから石を打つ座標を入力する */
Point readMove (int color) {
  int x, y;
  while (true) { // 適切な位置が選択されるまでループ
    System.out.print (“x?”);
    String inputString = keyBoardScanner.next();
    try {
      x = Integer.parseInt (inputString);
    } catch (NumberFormatException e) {  continue; // 整数以外 }
    if (place <1 || 8<place)  continue; // 範囲外
                                    :
```

33

## 手の入力

```
                                    :
   System.out.print (“y?”);
   try {
     y = Integer.parseInt (inputString);
   } catch (NumberFormatException e) {  continue; // 整数以外 }
   if (place <1 || 8<place)  continue; // 範囲外
   Point point = new Point (x, y);
   if (!isLegalMove (point, color)) continue; // 合法手ではない
   break;
  }
  return point;
}
```

34

## 手のランダム選択

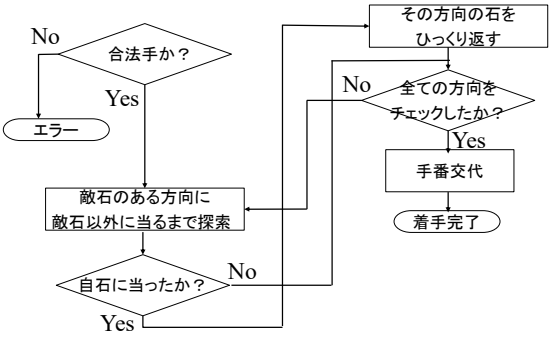
```
import java.util.Random;

/* 合法手からランダムに1つ選ぶ */
Point randomSelectMove (int color) {
  long seed = System.currentTimeMillis(); // 現在時刻を得る
  Random rnd = new Random (seed); // 乱数生成
  ArrayList<Point> moveList = generateMoveList (color);
  int r = rnd.nextInt (moveList.size()); // 合法手の数以下の乱数を得る
  Point point = moveList.get (r);
  return point;
}
```

35

## 着手の処理

選択したマスに対して以下の処理を行う



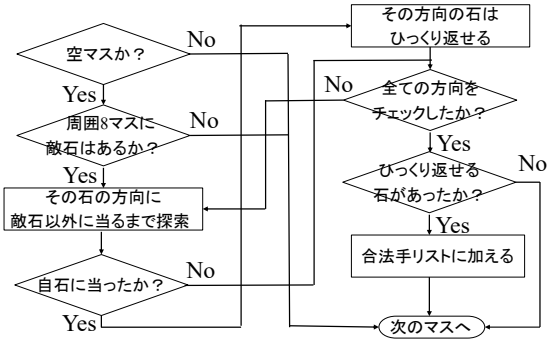

36

```
/* マス(x,y)に石を置くメソッド */
void move (Point point, int color) {
  int x = point.x, y=point.y;
  if (isLegalMove (x, y, color)) error(); // 合法手で無ければエラー
  board[x][y] = color; // 石を置く
  for (int i=0; i<8; ++i)
    if (board[x+vx[i]][y+vy[i]] == -color) { // 隣の石が敵石の場合
      int k=2;
      // 敵石以外に当るまで探索
      while (board[x+k*vx[i]][y+k*vy[i]] == -color) ++k;
      if (board[x+k*vx[i]][x+k*vy[i]] == color) // 自石に当った場合
        for (int m=1; m<k; ++m)
          board[x+m*vx[i]][y+m*vy[i]] = color; // 間の石を反転
    }
}
```

37

# 合法手の生成(再掲)

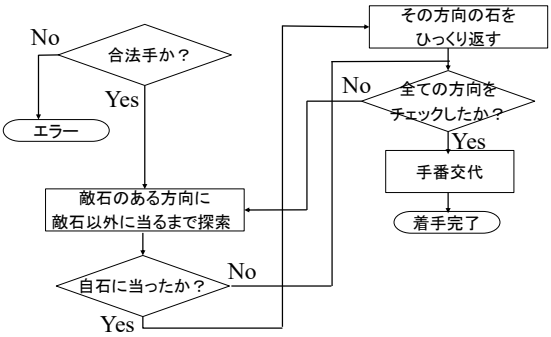
各マスに対して以下の処理を行う




38

# 着手の処理(再掲)

選択したマスに対して以下の処理を行う




39

# 合法手の判定と着手

- 合法手の判定と着手
  - 合法手の判定と着手は重なる部分が多い
  - 着手の前には必ず合法手の判定をしている

⇩

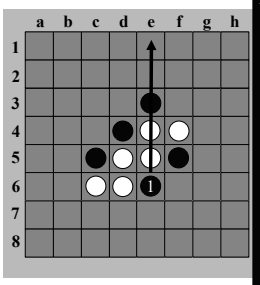
合法手の判定時に情報を残しておけば
着手時に利用できる

判定時に、合法手か否かだけではなく、
どちらの方向に引っくり返せる石があるかも判定しておく

40

# 合法手の判定と着手

上方向への探索




```
if (盤[x][y-1]==白) {
 /* 上方向に白石が続く限り探索 */
 for (v=-2; 盤[x][y+v]==白; --v));
 if (盤[x][y+v]==黒) {
   間の石をひっくり返せる
   上方向 = 1;
 } else { /* 空マスまたは壁の場合 */
   上方向の石はひっくり返せない
   上方向 = 0
 }
}
```

どちらの方向に石を引っ繰り返せるか記憶

41

# 合法手の判定と着手

- 8方向それぞれで石を引っくり返せるか記憶
  ⇒8ビットで表現できる

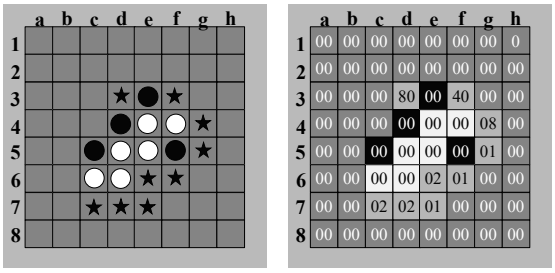
| 方向 | 左上 | 上 | 右上 | 左 | 右 | 左下 | 下 | 右下 |
|---|---|---|---|---|---|---|---|---|
| ビット | 01<br>00000001 | 02<br>00000010 | 04<br>00000100 | 08<br>00001000 | 10<br>00010000 | 20<br>00100000 | 40<br>01000000 | 80<br>10000000 |

例：上、左下、右下の3方向に引っくり返せる場合
02 + 20 + 80 = A2 (10100010)

どの方向にも引っ繰り返せない場合は 00 (00000000)

42

# 合法手リストの生成




各マスで引っくり返せる方向を記憶

43

```
/* マス(x,y)に石を置けるか判定するメソッド */
int isLegalMove (Point point, int color) {
  int x = point.x, y=point.y;
  dir = 0; // 方向ビット
  if (borad [x][y] != EMPTY) return false;
  for (int i=0; i<8; ++i)
     if (board[x+vx[i]][y+vy[i]] == -color) { // 隣の石が敵石の場合
        int k=2;
        // 敵石以外に当るまで探索
        while (board[x+k*vx[i]][y+k*vy[i]] == -color) ++k;
        if (board[x+k*vx[i]][x+k*vy[i]] == color) { // 自石に当った場合
           dir |= (1 << i); // 対応する方向ビットを1にする
        }
     }
  return dir:
}
```

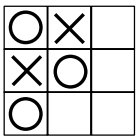
44

```
/* マス(x,y)に石を置くメソッド */
void move (Point point, int color) {
  int x = point.x, y=point.y;
  int dir = isLegalMove (x, y, color);
  if (dir == 0) error(); // 合法手が無ければエラー
  board[x][y] = color; // 石を置く
  for (int i=0; i<8; ++i)
     if ((dir  & (1 << i)) != 0) { // 引っくり返せる方向の場合
        int k=1;
        // 敵石以外(=自石)に当るまで探索
        while (board[x+k*vx[i]][y+k*vy[i]] == -color) {
          board[x+k*vx[i]][y+k*vy[i]] = color; // 敵石を反転
          ++k;
        }
     }
}
```

45

# 宿題:3目並べの着手選択

- 3目並べ着手選択
  - 先手後手それぞれを人間かCPUのどちらが受け持つかを選べるようにせよ
    - CPUは合法手からランダムに選択する



空きマスが4箇所あるので
0〜3の乱数を発生させて
着手選択

46

# 3目並べの実行例

```
% java tictactoe
○はCOMが持ちますか？ (Y/N) : n
×はCOMが持ちますか？ (Y/N) : y
 ┌─┬─┬─┐
 |7|8|9|
 ├─┼─┼─┤
 |4|5|6|
 ├─┼─┼─┤
 |1|2|3|
 └─┴─┴─┘
○の番です
打つ位置(1-9)を選んでください :
```

47