

SOLUTIONS

1. OCaml Higher-Order functions (12 points)

Recall the higher-order list processing functions `transform` and `fold` (reproduced in Appendix A). Each part of this problem below begins with a sample function written using simple recursion over lists, followed by several alternative versions written using `fold`. In each part, first indicate what the function returns for the sample input shown. Then mark all of the alternatives that implement the same behavior as the recursive sample. **There may be zero, one, or more than one such function.** Some of the alternatives do not typecheck—do not mark these.

(a)

```
let rec func1 (x:'a) (lst: 'a list) : int =
  begin match lst with
  | [] -> 0
  | hd :: tl -> if x = hd then 0 else 1 + func1 x tl
  end
let ans1 = func1 3 [0; 1; 2; 3]
```

ANSWER: `func1` calculates the index of the item of the list (or the length of the list if the item is not found). `ans1 = 4`.

Which of the following functions behave the same as `func1` on all inputs? (Check all that apply.)

- ☒

```
let func1 (x: 'a) (lst: 'a list) : int =
  fold (fun hd acc -> if (x = hd) then 0 else 1 + acc) 0 lst
```
- ☐

```
let func1 (x: 'a) (lst: 'a list) : int =
  fold (fun hd acc -> 1 + acc) 0 lst
```
- ☒

```
let func1 (x: 'a) (lst: 'a list) : int =
  fold (fun y acc -> if y then 0 else 1 + acc)
    0
  (transform (fun hd -> x = hd) lst)
```

(b)

```
let func2 (lst: 'a list): 'a list =
  let rec loop (res: 'a list) (acc: 'a list) =
    begin match res with
    | [] -> acc
    | hd :: tl -> loop tl (hd :: acc)
    end in
  loop lst []
```

```
let ans2 = func2 [0; 1; 2; 3]
```

ANSWER: `func2` reverses a list. `ans2 = [3; 2; 1; 0]`.

Which of the following functions behave the same as `func2` on all inputs? Recall that `@` appends two lists in OCaml. (Check all that apply.)

- ☐

```
let func2 (lst: 'a list): 'a list =
  fold (fun x acc -> x :: acc) [] lst
```
- ☒

```
let func2 (lst: 'a list) : 'a list =
  fold (fun x acc -> acc @ [x]) [] lst
```
- ☐

```
let func2 (lst: 'a list): 'a list =
  fold (fun x xs -> x @ xs) [] lst
```

2. OCaml queues and Java Linked Lists (13 points)

Consider the following OCaml functions that work with the queue data structure shown in Appendix B. The Java `LinkedList` class also implements a mutable, linked data structure. For each OCaml function below, use the documentation in Appendix C to determine which method of the `LinkedList` class provides the most similar functionality, or write none if there is no corresponding method in this class. This question tests your understanding of OCaml, so each function is called f .

For some of these operations, you may also be asked to indicate whether the function f is tail recursive.

(a) `let f (q: 'a queue) : bool =
 q.head = None`

Most similar `LinkedList` method:

(b) `let f (q: 'a queue) : unit =
 q.head <- None; q.tail <- None`

Most similar `LinkedList` method:

(c) `let f (q: 'a queue) : int =
 let rec loop (no: 'a qnode option) : int =
 begin match no with
 | None -> 0
 | Some n -> 1 + loop n.next
 end
 in loop q.head`

Most similar `LinkedList` method:

Is this function tail recursive? ☐ Yes ☒ No

(d) `let f (q: 'a queue) (elt:'a) : int =
 let rec loop (no: 'a qnode option) (i:int) : int =
 begin match no with
 | None -> -1
 | Some n -> if n.v = elt then i else loop n.next (i+1)
 end
 in loop q.head 0`

Most similar `LinkedList` method:

Is this function tail recursive? ☒ Yes ☐ No

(e) `let f (q: 'a queue) : 'a queue =
 let rec loop (no: 'a qnode option) (q2 : 'a queue) : 'a queue =
 begin match no with
 | None -> q2
 | Some n -> enq q2 n.v ; loop n.next q2
 end
 in let q2 = { head = None; tail = None}
 in loop q.head q2`

Most similar `LinkedList` method:

Is this function tail recursive? ☒ Yes ☐ No

3. TreeSet in Java (10 points)

The Java `TreeSet` class is implemented using a Binary Search Tree (BST). This class maintains the Binary Search Tree Invariant by storing the entries in the tree in order. Based on your understanding of BSTs in OCaml, which of the following methods of this class make use of this invariant?

(a) `int size()`

Returns the number of elements in this set (its cardinality).

Uses BST invariant: ☐ Yes ☒ No

(b) `boolean contains(Object value)`

Returns true if this set contains the specified element.

Uses BST invariant: ☒ Yes ☐ No

(c) `K floor(E e)`

Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.

Uses BST invariant: ☒ Yes ☐ No

(d) `boolean isEmpty()`

Returns true if this set contains no elements.

Uses BST invariant: ☐ Yes ☒ No

(e) `Iterator<E> iterator()`

Returns an iterator over the elements in this set in ascending order.

Uses BST invariant: ☒ Yes ☐ No

Connect Four

The remaining exam problems refer to a partial implementation of the game shown in Appendices D, E, F, and G and summarized below.

The *Connect Four* game features two players (`WHITE` and `BLACK`) taking turns adding pieces to a game board. In the physical version of the game, the game board is held upright, so players can only add their piece to the lowest empty spot in a given column. A player wins if they line up four of their pieces in a straight line, either vertically or horizontally. (For simplicity, this version does not look at diagonals.) The `WHITE` player goes first.

A sample game in progress is shown in the figure below on the left. By selecting the third column, the `BLACK` player can win by stacking four of their pieces in this column, as shown in the figure on the right.

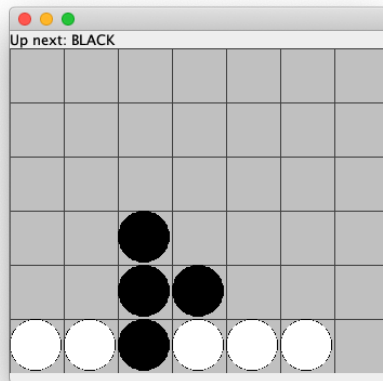


Figure 1: *Connect Four* game in progress.
It is `BLACK`'s turn.

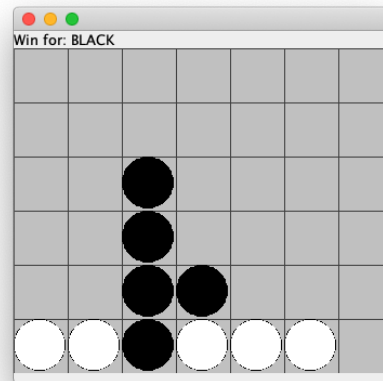


Figure 2: `BLACK` plays in the third column
to win.

Take a moment now to familiarize yourself with the code in the appendices. Appendix D defines classes that represent the two players. The code shown in Appendices E-G is all part of the same class, called `ConnectFour`, with the structure shown below.

```
class ConnectFour {
    public static final int ROWS = 6;
    public static final int COLUMNS = 7;

    // Appendix E: instance variables and methods for game logic
    // Appendix F: inner class View
    // Appendix G: instance variables for GUI, inner class Mouse,
    //              and ConnectFour constructor

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new ConnectFour());
    }
}
```

(There is nothing to answer on this page.)

4. Understanding OO Class Definitions (15 points)

The following questions refer to the classes defined in Appendix D that represent the two players in the *Connect Four* game and to the `LinkedList` class from the Collections Framework (Appendix C).

Which of the following code blocks is legal Java code that will not cause any compile-time (i.e. type checking) or run-time errors? If it is legal code, check the “Legal Code” box and answer the questions that follow it. If it is not legal, check one of the “Not Legal” options and explain why. You can assume each block below is independent and written in some static method defined in the `ConnectFour` class and that the appropriate imports have been made at the top of the file.

(a) (3 points)

```
Player p = Player.WHITE;
Player n = p.getNext();
boolean ans = (n == p);
```

☒ Legal Code

A. The static type of `p` is `Player`.

B. The dynamic class of `p` is `WhitePlayer`.

C. The value of `ans` is `false`.

☐ Not Legal — Will compile, but will throw an `Exception` when run

☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

(b) (3 points)

```
Player p = new Player(Color.WHITE);
boolean ans = p.equals(Player.WHITE);
```

☐ Legal Code

A. The static type of `p` is .

B. The dynamic class of `p` is .

C. The value of `ans` is .

☐ Not Legal — Will compile, but will throw an `Exception` when run

☒ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

Cannot make an instance of an abstract class

(c) (3 points)

```
Player p = new WhitePlayer();  
boolean ans = p.equals(Player.WHITE);
```

☒ Legal Code

- A. The static type of p is .
- B. The dynamic class of p is .
- C. The value of ans is .

- ☐ Not Legal — Will compile, but will throw an Exception when run
- ☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

(d) (3 points)

```
List<Player> lst = new LinkedList<Player>();  
lst.add(Player.WHITE);  
boolean ans = lst.contains(Player.WHITE);
```

☒ Legal Code

- A. The static type of lst is .
- B. The dynamic class of lst is .
- C. The value of ans is .

- ☐ Not Legal — Will compile, but will throw an Exception when run
- ☐ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

(e) (3 points)

```
List<WhitePlayer> lst = new LinkedList<WhitePlayer>();  
lst.add(Player.BLACK);  
boolean ans = lst.contains(Player.BLACK);
```

☐ Legal Code

- A. The static type of lst is .
- B. The dynamic class of lst is .
- C. The value of ans is .

- ☐ Not Legal — Will compile, but will throw an Exception when run
- ☒ Not Legal — Will not compile

Reason for not legal (in either of the two illegal cases above):

5. Invariants (10 points)

The state of the connect four game is stored by the following instance variables in the class `ConnectFour`.

```
private Player[][] board;    // the game board
private Player player;      // current player
private Player winner;      // null if no player has yet won
```

These instance variables are initialized by the `start` method, shown below.

```
public void start() {
    board = new Player[ROWS][COLUMNS];
    player = Player.WHITE;
    winner = null;
}
```

The `board` is represented by a two-dimensional array that stores the locations of the players' pieces, indexed by their row number and column number. The current `player` is either `Player.WHITE` or `Player.BLACK`. We also keep track of whether either player is the winner of the game.

For example, in the state shown in Figure 1 on page 5, `player` is equal to `Player.BLACK`, the game is still in progress so `winner` is `null`, and `board` is equal to the 2D array shown below. (For conciseness we write `Player.WHITE` as `WHITE` and `Player.BLACK` as `BLACK`).

```
new Player[][] {
    {null, null, null, null, null, null, null},
    {null, null, null, null, null, null, null},
    {null, null, null, null, null, null, null},
    {null, null, BLACK, null, null, null, null},
    {null, null, BLACK, BLACK, null, null, null},
    {WHITE, WHITE, BLACK, WHITE, WHITE, WHITE, null};
}
```

Locations on the board are referenced by `board[row][column]`. For example, position `board[5][0]` contains the `WHITE` player's piece in the last (lowest) row and first (leftmost) column.

Which of the properties below would make good invariants for the `ConnectFour` class? (*i.e.*, which properties should be true after the instance variables have been initialized and throughout the execution of the application?)

- (a) True ☒ False ☐ `board` is never `null`
- (b) True ☐ False ☒ `winner` is never `null`
- (c) True ☒ False ☐ `board[row]` is never `null` for any $0 \leq \text{row} < \text{ROWS}$.
- (d) True ☐ False ☒ `board[row][col]` is never `null` for any $0 \leq \text{col} < \text{COLUMNS}$ and $0 \leq \text{row} < \text{ROWS}$.
- (e) True ☒ False ☐ If `board[row][col]` is not `null`, then for any k such that $\text{row} < k < \text{ROWS}$, `board[k][col]` is not `null`.

6. 2D array Programming (15 points)

Complete the `addPiece` method of the `ConnectFour` class. This method should update the `board` with a new piece for the current `player` in the specified column. For example, if called when the `board` is equal to the sample board shown on page 8, and when the current player is `Player.BLACK`, this method should update `board[2][2]` to `Player.BLACK`.

The method should return whether the new piece was successfully added. In the case that the specified column is full, the method should return **false**. This method should only modify `board`—it should not update `player` or `winner`.

```
// update board with a new piece for the current player in the specified column
// returns false if the column is full
public boolean addPiece (int col) {
    for (int r=ROWS-1; r >= 0; r--) {
        if (board[r][col] == null) {
            board[r][col] = player;
            return true;
        }
    }
    return false;
}
```

Now think about the invariants that you checked “True” in the previous problem. Select **one** of these invariants and describe how, if it did not hold, your implementation of `addPiece` could behave incorrectly.

Selected invariant: (a) ☒ (b) ☐ (c) ☒ (d) ☐ (e) ☒

What could go wrong in `addPiece` if this invariant does not hold:

If (a) or (c) is not true, then the method could raise a `NullPointerException` when accessing the board. If (e) is not true, then some implementations of the method may not place the piece in the correct spot on the board (i.e. the lowest position in the column with a null value).

7. Iterators (15 points)

To check whether a player has won, we use the following methods to search for four pieces in a row either horizontally (rows) or vertically (columns).

```
// check all arrays produced by the iterator for win by player p
public static boolean checkAll(Iterator<Player[]> it, Player p) {
    while (it.hasNext()) {
        Player[] arr = it.next();
        if (checkFourInARow(arr, p)) {
            return true;
        }
    }
    return false;
}

// Check whether the current player has won the game
public boolean checkWin() {
    if (checkAll(new RowIterator(), player)) { return true; }
    return checkAll(new ColumnIterator(), player);
}
```

In the `checkWin` method, the classes `RowIterator` and `ColumnIterator` each implement the interface `Iterator<Player[]>`, providing access to a sequence of arrays representing the individual rows and columns respectively. For example, if the board is equal to the sample board on page 8, and the variable `cit` is a newly-created instance of the `ColumnIterator` class, then the following JUnit test will pass:

```
Player[] column1and2 = {null, null, null, null, null, WHITE};
Player[] column3 = {null, null, null, BLACK, BLACK, BLACK};

assertArrayEquals(cit.next(), column1and2);
assertArrayEquals(cit.next(), column1and2);
assertArrayEquals(cit.next(), column3);
```

For this problem you will complete the `ColumnIterator` class, an inner class of `ConnectFour`, on the next page. We have already declared and initialized the instance variable `currColumn`. You must complete the `hasNext` and `next` methods and their behavior must match the description given by the documentation. You may not define any additional class members (instance variables, constructors, or methods). HINT: because this class is an inner class it has access to the `board` instance variable of `ConnectFour`.

(Complete the code on the next page. There is nothing to answer on this page.)

```

class ColumnIterator implements Iterator<Player[]> {
    private int currColumn = 0;
    public boolean hasNext() {
        return currColumn < COLUMNS;
    }
    /** Returns the next element in the iteration.
     *
     * @throws NoSuchElementException if the iteration has no more elements
     */
    public Player[] next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        Player[] ret = new Player[ROWS];
        for (int i=0; i<ROWS; i++) {
            ret[i] = board[i][currColumn];
        }
        currColumn++;
        return ret;
    }
}

```

8. Java True/False (15 points)

The following questions refer to the `ConnectFour` code shown in Appendix E.

- (a) True ☐ False ☒ Line 58 is an example of the use of parametric polymorphism (i.e. generics).
The static type of `player` is `Player` and does not use generics.
- (b) True ☒ False ☐ Line 85 is an example of the use of dynamic dispatch. *The dynamic class of `it` determines which `hasNext` method will be called.*
- (c) True ☐ False ☒ Line 87 is an example of the use of dynamic dispatch. *The method `checkFourInARow` is static.*
- (d) True ☒ False ☐ Line 96 is an example of the use of subtype polymorphism. *The type `RowIterator` is a subtype of `Iterator<Piece[]>`*
- (e) True ☒ False ☐ Line 96 is an example of the use of parametric polymorphism (i.e. generics).
The type `Iterator<Piece[]>` uses generics

The following questions refer to the `ConnectFour` code shown in Appendix F.

- (f) True ☒ False ☐ The type `View` is a subtype of `Object`.
- (g) True ☐ False ☒ The class `View` is a subclass of `ConnectFour`. *View is an innerclass not a subclass*
- (h) True ☐ False ☒ The methods and constructors in class `View` may refer to the private instance variable `color` of class `Player` (Appendix D). *Private instance variables can only be accessed in the same class or inner classes.*
- (i) True ☒ False ☐ The call to `super.paintComponent` on line 117 refers to a member of class `JPanel`. *The superclass of `View` is `JPanel`.*

The following questions refer to the `ConnectFour` code shown in Appendix G and to the classes of the Java Swing library. For reference, documentation for the Swing library appears in Appendix H.

- (j) True ☐ False ☒ The class `Mouse` is a subclass of `JPanel`. *Mouse extends `MouseAdapter`*
- (k) True ☒ False ☐ The type `Mouse` is a subtype of `MouseListener`. *Mouse implements `MouseListener`*
- (l) True ☒ False ☐ The type `MouseListener` is a subtype of `Object`. *All interfaces are subtypes of `Object`*

- (m) True ☒ False ☐ The class `Mouse` inherits a method called `mousePressed`. *The superclass `MouseAdapter` includes this method, so it is inherited by `Mouse`*
- (n) True ☒ False ☐ The methods and constructors in class `Mouse` may refer to the private instance variables of class `ConnectFour`. *`Mouse` is an inner class of `ConnectFour`*
- (o) True ☐ False ☒ We can remove line 149 in Appendix G and the application would still work. *The application needs to indicate that the view should be repainted whenever the model changes. Otherwise Swing will not redisplay the UI element.*

9. Exceptions and ActionListeners (15 points)

Suppose we would like to add a “save” button to the *Connect Four* game. Pressing this button should record the current state of the game to a save file using the following method to be added in class `ConnectFour` (the implementation of this method is not shown). If the game cannot be saved, this method throws an `IOException`. In this case, the application should use the `info` label to display an error message to the user.

```
// write the game state to a file
// throws IOException if the file cannot be written to
public void saveGame() throws IOException {
    // not shown
}
```

To implement the save button, we’ll add the following lines to Appendix G at at line 178.

```
JButton save = new JButton("save");
panel.add(save, BorderLayout.PAGE_END);
```

What should come next? Which of the following code blocks correctly add the action listener for the save button. Check **Yes** if the code would work, or **No** if it either would not compile or would not implement the desired behavior. There may be zero, one or more correct code blocks.

(a) ☒ Yes ☐ No

```
save.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            saveGame();
        } catch (IOException io) {
            info.setText("Cannot save game");
        }
    }
});
```

(b) ☐ Yes ☒ No

```
save.addActionListener(
    try {
        new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) throws IOException {
                saveGame();
            }
        }
    } catch (IOException io) {
        info.setText("Cannot save game.");
    });
```

(c) ☒ Yes ☐ No

```
save.addActionListener((ActionEvent e) -> {  
    try {  
        saveGame();  
    } catch (IOException io) {  
        info.setText("Cannot save game");  
    }  
});
```

(d) ☐ Yes ☒ No

```
try {  
    save.addActionListener(new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            saveGame();  
        }  
    });  
} catch (IOException io) {  
    info.setText("Cannot save game.");  
}
```

(e) ☐ Yes ☒ No

```
save.addActionListener((ActionEvent e) -> {  
    saveGame();  
    throw new IOException("Cannot save game.");  
});
```

CIS 120 Final Exam — Appendices

A OCaml Higher-Order Functions

```
let rec transform (f: 'a -> 'b) (lst : 'a list) : 'b list =
  begin match lst with
  | [] -> []
  | hd :: tl -> f hd :: transform f tl
  end

let rec fold (acc: 'a -> 'b -> 'b) (base:'b) (lst:'a list) : 'b =
  begin match lst with
  | [] -> base
  | hd :: tl -> acc hd (fold acc base tl)
  end
```

B OCaml Queue Code

```
type 'a qnode = { v: 'a;
                  mutable next: 'a qnode option }

type 'a queue = { mutable head: 'a qnode option;
                  mutable tail: 'a qnode option }

(* Add an element to the tail of a queue *)
let enq (q: 'a queue) (elt: 'a) : unit =
  let newnode = Some { v = elt; next = None } in
  begin match q.tail with
  | None ->
    q.head <- newnode;
    q.tail <- newnode
  | Some n ->
    n.next <- newnode;
    q.tail <- newnode
  end

(* Remove an element from the head of the queue *)
let deq (q: 'a queue) : 'a =
  begin match q.head with
  | None ->
    failwith "deq called on empty queue"
  | Some n ->
    q.head <- n.next;
    begin match n.next with
    | None -> q.tail <- None; n.v
    | Some _ -> n.v
    end
  end
```


C Java Docs: `class LinkedList<E> implements List<E>` (excerpt)

boolean `add(E e)`

Appends the specified element to the end of this list.

void `clear()`

Removes all of the elements from this list. The list will be empty after this call returns.

`Object clone()`

Returns a shallow copy of this `LinkedList`. (The elements themselves are not cloned.)

public boolean `contains(Object o)`

Returns **true** if this list contains the specified element.

public E `get(int index)`

Returns the element at the specified position in this list.

public E `getFirst()`

Returns the first element in this list.

public int `indexOf(Object o)`

Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

boolean `isEmpty()`

Returns **true** if the list contains no elements.

`Iterator<E> iterator()`

Returns an iterator over the elements in this list in proper sequence.

`E removeLast()`

Removes and returns the last element from this list.

int `size()`

Returns the number of elements in this list.

D ConnectFour: Player classes

The following class declarations are in the same package as the `ConnectFour` class and are used to represent the two players and their pieces in the game. The `draw` method is implemented in the player class but not shown.

```
1  abstract class Player {
2      public static final Player WHITE = new WhitePlayer();
3      public static final Player BLACK = new BlackPlayer();
4
5      private final Color color;
6
7      public Player (Color c) {
8          this.color = c;
9      }
10
11     // what player goes after this one
12     public abstract Player getNext();
13
14     // draw a player's piece at location x, y using the player's color
15     public void draw(Graphics gc, int x, int y) {
16         // not shown
17     }
18 }
19
20 class BlackPlayer extends Player {
21     public BlackPlayer() {
22         super(Color.BLACK);
23     }
24     @Override
25     public String toString() {
26         return "BLACK";
27     }
28     @Override
29     public Player getNext() {
30         return Player.WHITE;
31     }
32 }
33
34 class WhitePlayer extends Player {
35     public WhitePlayer () {
36         super(Color.WHITE);
37     }
38     @Override
39     public String toString() {
40         return "WHITE";
41     }
42     @Override
43     public Player getNext() {
44         return Player.BLACK;
45     }
46 }
```

E ConnectFour: Game Logic (Model)

The following declarations are members of class `ConnectFour`, defined on page 5, and are related to the game logic. Some of the code below has been omitted.

```
51     private Player[][] board;    // the game board
52     private Player player;      // current player
53     private Player winner;      // null if no player has yet won
54
55     // initialize the game state
56     public void start () {
57         board = new Player[ROWS][COLUMNS];
58         player = Player.WHITE;
59         winner = null;
60     }
61
62     // update game board as current player takes their turn
63     // returns false if the specified column is full
64     public boolean addPiece (int column) {
65         // not shown
66     }
67
68     // iterate through the rows of the game board from top to bottom
69     class RowIterator implements Iterator<Player[]> {
70         // not shown
71     }
72
73     // iterate through the columns of the game board from left to right
74     class ColumnIterator implements Iterator<Player[]> {
75         // not shown
76     }
77
78     // check whether an array contains four contiguous pieces of player p
79     public static boolean checkFourInARow(Player[] array, Player p) {
80         // not shown
81     }
82
83     // check all arrays produced by the iterator for win by player p
84     public static boolean checkAll(Iterator<Player[]> it, Player p) {
85         while (it.hasNext()) {
86             Player[] arr = it.next();
87             if (checkFourInARow(arr, p)) {
88                 return true;
89             }
90         }
91         return false;
92     }
93
94     // check whether the current player has won the game
95     public boolean checkWin() {
96         if (checkAll(new RowIterator())) { return true; }
97         return checkAll(new ColumnIterator());
98     }
```

F ConnectFour:View inner class

The following declarations are members of class `ConnectFour`, defined on page 5, and are related to the display of the game. Some of the code below has been omitted.

```

99      // inner class of ConnectFour
100     //
101     public class View extends JPanel {
102
103         public View() {
104             setBackground(Color.WHITE);
105         }
106
107         // width of columns and height of rows, in pixels
108         public static final int SIZE = 50;
109
110         // draw the grid lines
111         private void drawGrid(Graphics gc) {
112             // not shown
113         }
114
115         @Override
116         public void paintComponent(Graphics gc) {
117             super.paintComponent(gc);
118             drawGrid(gc);
119             for (int row = 0; row < ROWS; row++) {
120                 for (int col = 0; col < COLUMNS; col++) {
121                     if (board[row][col] != null) {
122                         board[row][col].draw(gc, row * SIZE, col * SIZE);
123                     }
124                 }
125             }
126         }
127
128         @Override
129         public Dimension getPreferredSize() {
130             return new Dimension(COLUMNS * SIZE, ROWS * SIZE);
131         }
132     }
133 }
```

G ConnectFour:Mouse inner class and Constructor

The following declarations are members of class `ConnectFour`, defined on page 5, and are related to the GUI and its interaction.

```
134     // instance variables for the game GUI
135     //
136     private JLabel info = new JLabel();    // status text at top of window
137     private View view = new View();       // display of game board
138
139
140     class Mouse extends MouseAdapter implements MouseListener {
141         @Override
142         public void mouseClicked(MouseEvent e) {
143             Point p = e.getPoint();
144             int col = p.x / View.SIZE;
145             if (winner != null) {
146                 return; // game is over, no more moves allowed
147             }
148             if (addPiece(col)) {
149                 view.repaint();
150                 if (checkWin()) {
151                     winner = player;
152                     info.setText("Win for: " + winner);
153                 } else {
154                     player = player.next();
155                     info.setText("Up next: " + player);
156                 }
157             } else {
158                 info.setText("Cannot play there!");
159             }
160         }
161     }
162
163     // constructor
164     public ConnectFour() {
165         final JFrame frame = new JFrame();
166         JPanel panel = new JPanel();
167         frame.setContentPane(panel);
168         panel.setLayout(new BorderLayout());
169
170         panel.add(view, BorderLayout.CENTER);
171         panel.add(info, BorderLayout.PAGE_START);
172
173         start();
174         info.setText("Up next: " + player);
175
176         MouseListener m = new Mouse();
177         view.addMouseListener(m);
178
179         frame.pack();
180         frame.setVisible(true);
181         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
182     }
```

H Swing library documentation

H.1 Java Docs: `MouseAdapter` class (excerpt)

```
class MouseAdapter implements MouseListener

void mouseClicked(MouseEvent e)
void mouseEntered(MouseEvent e)
void mouseExited(MouseEvent e)
void mousePressed(MouseEvent e)
void mouseReleased(MouseEvent e)
```

H.2 Java Docs: `JPanel` class (excerpt)

```
class JPanel extends JComponent

Component add(Component comp)
void addMouseListener(MouseListener l)
Dimension getPreferredSize()
void paintComponent(Graphics g)
void repaint()
void setBackground(Color bg)
```

H.3 Java Docs: `JButton` class (excerpt)

```
class JButton extends JComponent

JButton(String text)
void addActionListener(ActionListener l)
Dimension getPreferredSize()
void paintComponent(Graphics g)
void repaint()
```

H.4 Java Docs: `ActionListener` interface (excerpt)

```
interface ActionListener extends EventListener

void actionPerformed(ActionEvent e)
```