```
22          throw new java.lang.IndexOutOfBoundsException("...");
23       }
24       return items[convert(i)];
25    }
26
27    public void set(int i, T item) {
28       items[convert(i)] = item;
29    }
30 }
```

There are a number of things here which are easy to make mistakes on, such as:

- In Java, we cannot create an array of the generic type. Instead, we must either cast the array or define `items` to be of type `List<T>`. For simplicity, we have done the former.
- The % operator will return a negative value when we do `negValue % posVal`. For example, `-8 % 3` is `-2`. This is different from how mathematicians would define the modulus function. We must add `items.length` to a negative index to get the correct positive result.
- We need to be sure to consistently convert the raw index to the rotated index. For this reason, we have implemented a `convert` function that is used by other methods. Even the `rotate` function uses `convert`. This is a good example of code reuse.

Now that we have the basic code for `CircularArray` out of the way, we can focus on implementing an iterator.

Implementing the Iterator Interface

The second part of this question asks us to implement the `CircularArray` class such that we can do the following:

```
1  CircularArray<String> array = ...
2  for (String s : array) { ... }
```

Implementing this requires implementing the `Iterator` interface. The details of this implementation apply to Java, but similar things can be implemented in other languages.

To implement the `Iterator` interface, we need to do the following:

- Modify the `CircularArray<T>` definition to add `implements Iterable<T>`. This will also require us to add an `iterator()` method to `CircularArray<T>`.
- Create a `CircularArrayIterator<T>` which implements `Iterator<T>`. This will also require us to implement, in the `CircularArrayIterator`, the methods `hasNext()`, `next()`, and `remove()`.

Once we've done the above items, the for loop will "magically" work.

In the code below, we have removed the aspects of `CircularArray` which were identical to the earlier implementation.

```
1  public class CircularArray<T> implements Iterable<T> {
2     ...
3     public Iterator<T> iterator() {
4        return new CircularArrayIterator<T>(this);
5     }
6
7     private class CircularArrayIterator<TI> implements Iterator<TI>{
8        /* current reflects the offset from the rotated head, not from the actual
9         * start of the raw array. */
10       private int _current = -1;
```

```
11      private TI[] _items;
12
13      public CircularArrayIterator(CircularArray<TI> array){
14         _items = array.items;
15      }
16
17      @Override
18      public boolean hasNext() {
19         return _current < items.length - 1;
20      }
21
22      @Override
23      public TI next() {
24         _current++;
25         TI item = (TI) _items[convert(_current)];
26         return item;
27      }
28
29      @Override
30      public void remove() {
31         throw new UnsupportedOperationException("...");
32      }
33   }
34 }
```

In the above code, note that the first iteration of the for loop will call `hasNext()` and then `next()`. Be very sure that your implementation will return the correct values here.

When you get a problem like this one in an interview, there's a good chance you don't remember exactly what the various methods and interfaces are called. In this case, work through the problem as well as you can. If you can reason out what sorts of methods one might need, that alone will show a good degree of competency.

7.10 **Minesweeper:** Design and implement a text-based Minesweeper game. Minesweeper is the classic single-player computer game where an NxN grid has B mines (or bombs) hidden across the grid. The remaining cells are either blank or have a number behind them. The numbers reflect the number of bombs in the surrounding eight cells. The user then uncovers a cell. If it is a bomb, the player loses. If it is a number, the number is exposed. If it is a blank cell, this cell and all adjacent blank cells (up to and including the surrounding numeric cells) are exposed. The player wins when all non-bomb cells are exposed. The player can also flag certain places as potential bombs. This doesn't affect game play, other than to block the user from accidentally clicking a cell that is thought to have a bomb. (Tip for the reader: if you're not familiar with this game, please play a few rounds online first.)

This is a fully exposed board with 3 bombs. This is not shown to the user.

	1	1	1			
	1	*	1			
	2	2	2			
	1	*	1			
	1	1	1			
			1	1	1	
			1	*	1	

The player initially sees a board with nothing exposed.

?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

Clicking on cell (row = 1, col = 0) would expose this:

	1	?	?	?	?	?
	1	?	?	?	?	?
	2	?	?	?	?	?
	1	?	?	?	?	?
	1	1	1	?	?	?
			1	?	?	?
			1	?	?	?

The user wins when everything other than bombs has been exposed.

	1	1	1			
	1	?	1			
	2	2	2			
	1	?	1			
	1	1	1			
			1	1	1	
			1	?	1	

pg 129

SOLUTION

Writing an entire game—even a text-based one—would take far longer than the allotted time you have in an interview. This doesn't mean that it's not fair game as a question. It just means that your interviewer's expectation will not be that you actually write all of this in an interview. It also means that you need to focus on getting the key ideas—or structure—out.

Let's start with what the classes are. We certainly want a `Cell` class as well as a `Board` class. We also probably want to have a `Game` class.

We could potentially merge `Board` and `Game` together, but it's probably best to keep them separate. Err towards more organization, not less. `Board` can hold the list of `Cell` objects and do some basic moves with flipping over cells. `Game` will hold the game state and handle user input.

Design: Cell

`Cell` will need to have knowledge of whether it's a bomb, a number, or a blank. We could potentially subclass `Cell` to hold this data, but I'm not sure that offers us much benefit.

We could also have an enum `TYPE {BOMB, NUMBER, BLANK}` to describe the type of cell. We've chosen not to do this because `BLANK` is really a type of `NUMBER` cell, where the number is 0. It's sufficient to just have an `isBomb` flag.

It's okay to have made different choices here. These aren't the only good choices. Explain the choices you make and their tradeoffs with your interviewer.

We also need to store state for whether the cell is exposed or not. We probably do not want to subclass `Cell` for `ExposedCell` and `UnexposedCell`. This is a bad idea because `Board` holds a reference to the cells, and we'd have to change the reference when we flip a cell. And then what if other objects reference the instance of `Cell`?

It's better to just have a boolean flag for `isExposed`. We'll do a similar thing for `isGuess`.

```
1  public class Cell {
2    private int row;
3    private int column;
4    private boolean isBomb;
5    private int number;
6    private boolean isExposed = false;
7    private boolean isGuess = false;
8
9    public Cell(int r, int c) { ... }
10
11   /* Getters and setters for above variables. */
12   ...
13
14   public boolean flip() {
15     isExposed = true;
16     return !isBomb;
17   }
18
19   public boolean toggleGuess() {
20     if (!isExposed) {
21       isGuess = !isGuess;
22     }
23     return isGuess;
24   }
25
26   /* Full code can be found in downloadable code solutions. */
27 }
```

Design: Board

`Board` will need to have an array of all the `Cell` objects. A two-dimension array will work just fine.

We'll probably want `Board` to keep state of how many unexposed cells there are. We'll track this as we go, so we don't have to continuously count it.

`Board` will also handle some of the basic algorithms:

- Initializing the board and laying out the bombs.
- Flipping a cell.

- Expanding blank areas.

It will receive the game plays from the Game object and carry them out. It will then need to return the result of the play, which could be any of {clicked a bomb and lost, clicked out of bounds, clicked an already exposed area, clicked a blank area and still playing, clicked a blank area and won, clicked a number and won}. This is really two different items that need to be returned: successful (whether or not the play was successfully made) and a game state (won, lost, playing). We'll use an additional `GamePlayResult` to return this data.

We'll also use a `GamePlay` class to hold the move that the player plays. We need to use a row, column, and then a flag to indicate whether this was an actual flip or the user was just marking this as a "guess" at a possible bomb.

The basic skeleton of this class might look something like this:

```
1  public class Board {
2     private int nRows;
3     private int nColumns;
4     private int nBombs = 0;
5     private Cell[][] cells;
6     private Cell[] bombs;
7     private int numUnexposedRemaining;
8
9     public Board(int r, int c, int b) { ... }
10
11    private void initializeBoard() { ... }
12    private boolean flipCell(Cell cell) { ... }
13    public void expandBlank(Cell cell) { ... }
14    public UserPlayResult playFlip(UserPlay play) { ... }
15    public int getNumRemaining() { return numUnexposedRemaining; }
16 }
17
18 public class UserPlay {
19    private int row;
20    private int column;
21    private boolean isGuess;
22    /* constructor, getters, setters. */
23 }
24
25 public class UserPlayResult {
26    private boolean successful;
27    private Game.GameState resultingState;
28    /* constructor, getters, setters. */
29 }
```

Design: Game

The Game class will store references to the board and hold the game state. It also takes the user input and sends it off to Board.

```
1  public class Game {
2     public enum GameState { WON, LOST, RUNNING }
3
4     private Board board;
5     private int rows;
6     private int columns;
7     private int bombs;
8     private GameState state;
```

```
9
10    public Game(int r, int c, int b) { ... }
11
12    public boolean initialize() { ... }
13    public boolean start() { ... }
14    private boolean playGame() { ... } // Loops until game is over.
15 }
```

Algorithms

This is the basic object-oriented design in our code. Our interviewer might ask us now to implement a few of the most interesting algorithms.

In this case, the three interesting algorithms is the initialization (placing the bombs randomly), setting the values of the numbered cells, and expanding the blank region.

Placing the Bombs

To place the bombs, we could randomly pick a cell and then place a bomb if it's still available, and otherwise pick a different location for it. The problem with this is that if there are a lot of bombs, it could get very slow. We could end up in a situation where we repeatedly pick cells with bombs.

To get around this, we could take an approach similar to the card deck shuffling problem (pg 531). We could place the K bombs in the first K cells and then shuffle all the cells around.

Shuffling an array operates by iterating through the array from `i = 0` through `N-1`. For each `i`, we pick a random index between `i` and `N-1` and swap it with that index.

To shuffle a grid, we do a very similar thing, just converting the index into a row and column location.

```
1  void shuffleBoard() {
2    int nCells = nRows * nColumns;
3    Random random = new Random();
4    for (int index1 = 0; index1 < nCells; index1++) {
5      int index2 = index1 + random.nextInt(nCells - index1);
6      if (index1 != index2) {
7        /* Get cell at index1. */
8        int row1 = index1 / nColumns;
9        int column1 = (index1 - row1 * nColumns) % nColumns;
10       Cell cell1 = cells[row1][column1];
11
12       /* Get cell at index2. */
13       int row2 = index2 / nColumns;
14       int column2 = (index2 - row2 * nColumns) % nColumns;
15       Cell cell2 = cells[row2][column2];
16
17       /* Swap. */
18       cells[row1][column1] = cell2;
19       cell2.setRowAndColumn(row1, column1);
20       cells[row2][column2] = cell1;
21       cell1.setRowAndColumn(row2, column2);
22     }
23   }
24 }
```

Setting the Numbered Cells

Once the bombs have been placed, we need to set the values of the numbered cells. We could go through each cell and check how many bombs are around it. This would work, but it's actually a bit slower than is necessary.

Instead, we can go to each bomb and increment each cell around it. For example, cells with 3 bombs will get `incrementNumber` called three times on them and will wind up with a number of 3.

```
1  /* Set the cells around the bombs to the right number. Although the bombs have
2   * been shuffled, the reference in the bombs array is still to same object. */
3  void setNumberedCells() {
4    int[][] deltas = { // Offsets of 8 surrounding cells
5         {-1, -1}, {-1, 0}, {-1, 1},
6         { 0, -1},          { 0, 1},
7         { 1, -1}, { 1, 0}, { 1, 1}
8    };
9    for (Cell bomb : bombs) {
10     int row = bomb.getRow();
11     int col = bomb.getColumn();
12     for (int[] delta : deltas) {
13       int r = row + delta[0];
14       int c = col + delta[1];
15       if (inBounds(r, c)) {
16         cells[r][c].incrementNumber();
17       }
18     }
19   }
20 }
```

Expanding a Blank Region

Expanding the blank region could be done either iteratively or recursively. We implemented it iteratively.

You can think about this algorithm like this: each blank cell is surrounded by either blank cells or numbered cells (never a bomb). All need to be flipped. But, if you're flipping a blank cell, you also need to add the blank cells to a queue, to flip their neighboring cells.

```
1  void expandBlank(Cell cell) {
2    int[][] deltas = {
3         {-1, -1}, {-1, 0}, {-1, 1},
4         { 0, -1},          { 0, 1},
5         { 1, -1}, { 1, 0}, { 1, 1}
6    };
7
8    Queue<Cell> toExplore = new LinkedList<Cell>();
9    toExplore.add(cell);
10
11   while (!toExplore.isEmpty()) {
12     Cell current = toExplore.remove();
13
14     for (int[] delta : deltas) {
15       int r = current.getRow() + delta[0];
16       int c = current.getColumn() + delta[1];
17
18       if (inBounds(r, c)) {
19         Cell neighbor = cells[r][c];
20         if (flipCell(neighbor) && neighbor.isBlank()) {
21           toExplore.add(neighbor);
```

```
            }
        }
    }
}
```

You could instead implement this algorithm recursively. In this algorithm, rather than adding the cell to a queue, you would make a recursive call.

Your implementation of these algorithms could vary substantially depending on your class design.

7.11 File System: Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

pg 129

SOLUTION

Many candidates may see this problem and instantly panic. A file system seems so low level!

However, there's no need to panic. If we think through the components of a file system, we can tackle this problem just like any other object-oriented design question.

A file system, in its most simplistic version, consists of `Files` and `Directories`. Each `Directory` contains a set of `Files` and `Directories`. Since `Files` and `Directories` share so many characteristics, we've implemented them such that they inherit from the same class, `Entry`.

```
1  public abstract class Entry {
2    protected Directory parent;
3    protected long created;
4    protected long lastUpdated;
5    protected long lastAccessed;
6    protected String name;
7
8    public Entry(String n, Directory p) {
9      name = n;
10     parent = p;
11     created = System.currentTimeMillis();
12     lastUpdated = System.currentTimeMillis();
13     lastAccessed = System.currentTimeMillis();
14   }
15
16   public boolean delete() {
17     if (parent == null) return false;
18     return parent.deleteEntry(this);
19   }
20
21   public abstract int size();
22
23   public String getFullPath() {
24     if (parent == null) return name;
25     else return parent.getFullPath() + "/" + name;
26   }
27
28   /* Getters and setters. */
29   public long getCreationTime() { return created; }
30   public long getLastUpdatedTime() { return lastUpdated; }
31   public long getLastAccessedTime() { return lastAccessed; }
```

```
32    public void changeName(String n) { name = n; }
33    public String getName() { return name; }
34  }
35
36  public class File extends Entry {
37    private String content;
38    private int size;
39
40    public File(String n, Directory p, int sz) {
41      super(n, p);
42      size = sz;
43    }
44
45    public int size() { return size; }
46    public String getContents() { return content; }
47    public void setContents(String c) { content = c; }
48  }
49
50  public class Directory extends Entry {
51    protected ArrayList<Entry> contents;
52
53    public Directory(String n, Directory p) {
54      super(n, p);
55      contents = new ArrayList<Entry>();
56    }
57
58    public int size() {
59      int size = 0;
60      for (Entry e : contents) {
61        size += e.size();
62      }
63      return size;
64    }
65
66    public int numberOfFiles() {
67      int count = 0;
68      for (Entry e : contents) {
69        if (e instanceof Directory) {
70          count++; // Directory counts as a file
71          Directory d = (Directory) e;
72          count += d.numberOfFiles();
73        } else if (e instanceof File) {
74          count++;
75        }
76      }
77      return count;
78    }
79
80    public boolean deleteEntry(Entry entry) {
81      return contents.remove(entry);
82    }
83
84    public void addEntry(Entry entry) {
85      contents.add(entry);
86    }
87
```

```
88    protected ArrayList<Entry> getContents() { return contents; }
89  }
```

Alternatively, we could have implemented `Directory` such that it contains separate lists for files and subdirectories. This makes the `numberOfFiles()` method a bit cleaner, since it doesn't need to use the `instanceof` operator, but it does prohibit us from cleanly sorting files and directories by dates or names.

7.12 Hash Table: Design and implement a hash table which uses chaining (linked lists) to handle collisions.

pg 129

SOLUTION

Suppose we are implementing a hash table that looks like `Hash<K, V>`. That is, the hash table maps from objects of type K to objects of type V.

At first, we might think our data structure would look something like this:

```
1  class Hash<K, V> {
2    LinkedList<V>[] items;
3    public void put(K key, V value) { ... }
4    public V get(K key) { ... }
5  }
```

Note that `items` is an array of linked lists, where `items[i]` is a linked list of all objects with keys that map to index `i` (that is, all the objects that collided at `i`).

This would seem to work until we think more deeply about collisions.

Suppose we have a very simple hash function that uses the string length.

```
1  int hashCodeOfKey(K key) {
2    return key.toString().length() % items.length;
3  }
```

The keys `jim` and `bob` will map to the same index in the array, even though they are different keys. We need to search through the linked list to find the actual object that corresponds to these keys. But how would we do that? All we've stored in the linked list is the value, not the original key.

This is why we need to store both the value and the original key.

One way to do that is to create another object called `Cell` which pairs keys and values. With this implementation, our linked list is of type `Cell`.

The code below uses this implementation.

```
1  public class Hasher<K, V> {
2    /* Linked list node class. Used only within hash table. No one else should get
3     * access to this. Implemented as doubly linked list. */
4    private static class LinkedListNode<K, V> {
5      public LinkedListNode<K, V> next;
6      public LinkedListNode<K, V> prev;
7      public K key;
8      public V value;
9      public LinkedListNode(K k, V v) {
10       key = k;
11       value = v;
12     }
13   }
14
```