	FizzBuzz	Fizz	Buzz	Number
current % 3 == 0	true	true	false	false
current % 5 == 0	true	false	true	false
to print	FizzBuzz	Fizz	Buzz	current

For the most part, this can be handled by taking in "target" parameters and the value to print. The output for the `Number` thread needs to be overwritten, though, as it's not a simple, fixed string.

We can implement a `FizzBuzzThread` class which handles most of this. A `NumberThread` class can extend `FizzBuzzThread` and override the `print` method.

```
1  Thread[] threads = {new FizzBuzzThread(true, true, n, "FizzBuzz"),
2                      new FizzBuzzThread(true, false, n, "Fizz"),
3                      new FizzBuzzThread(false, true, n, "Buzz"),
4                      new NumberThread(false, false, n)};
5  for (Thread thread : threads) {
6    thread.start();
7  }
8
9  public class FizzBuzzThread extends Thread {
10   private static Object lock = new Object();
11   protected static int current = 1;
12   private int max;
13   private boolean div3, div5;
14   private String toPrint;
15
16   public FizzBuzzThread(boolean div3, boolean div5, int max, String toPrint) {
17     this.div3 = div3;
18     this.div5 = div5;
19     this.max = max;
20     this.toPrint = toPrint;
21   }
22
23   public void print() {
24     System.out.println(toPrint);
25   }
26
27   public void run() {
28     while (true) {
29       synchronized (lock) {
30         if (current > max) {
31           return;
32         }
33
34         if ((current % 3 == 0) == div3 &&
35             (current % 5 == 0) == div5) {
36           print();
37           current++;
38         }
39       }
40     }
41   }
42 }
43
44 public class NumberThread extends FizzBuzzThread {
```

```
45    public NumberThread(boolean div3, boolean div5, int max) {
46       super(div3, div5, max, null);
47    }
48
49    public void print() {
50       System.out.println(current);
51    }
52 }
```

Observe that we need to put the comparison of `current` and `max` before the if statement, to ensure the value will only get printed when `current` is less than or equal to `max`.

Alternatively, if we're working in a language which supports this (Java 8 and many other languages do), we can pass in a `validate` method and a `print` method as parameters.

```
1  int n = 100;
2  Thread[] threads = {
3     new FBThread(i -> i % 3 == 0 && i % 5 == 0, i -> "FizzBuzz", n),
4     new FBThread(i -> i % 3 == 0 && i % 5 != 0, i -> "Fizz", n),
5     new FBThread(i -> i % 3 != 0 && i % 5 == 0, i -> "Buzz", n),
6     new FBThread(i -> i % 3 != 0 && i % 5 != 0, i -> Integer.toString(i), n)};
7  for (Thread thread : threads) {
8     thread.start();
9  }
10
11 public class FBThread extends Thread {
12    private static Object lock = new Object();
13    protected static int current = 1;
14    private int max;
15    private Predicate<Integer> validate;
16    private Function<Integer, String> printer;
17    int x = 1;
18
19    public FBThread(Predicate<Integer> validate,
20                    Function<Integer, String> printer, int max) {
21       this.validate = validate;
22       this.printer = printer;
23       this.max = max;
24    }
25
26    public void run() {
27       while (true) {
28          synchronized (lock) {
29             if (current > max) {
30                return;
31             }
32             if (validate.test(current)) {
33                System.out.println(printer.apply(current));
34                current++;
35             }
36          }
37       }
38    }
39 }
```

There are of course many other ways of implementing this as well.

16

Solutions to Moderate

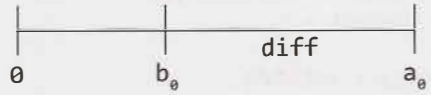
16.1 Number Swapper: Write a function to swap a number in place (that is, without temporary variables).

pg 181

SOLUTION

This is a classic interview problem, and it's a reasonably straightforward one. We'll walk through this using a_0 to indicate the original value of a and b_0 to indicate the original value of b. We'll also use `diff` to indicate the value of a_0 - b_0.

Let's picture these on a number line for the case where a > b.



First, we briefly set a to `diff`, which is the right side of the above number line. Then, when we add b and `diff` (and store that value in b), we get a_0. We now have b = a_0 and a = `diff`. All that's left to do is to set a equal to a_0 - `diff`, which is just b - a.

The code below implements this.

```
1  // Example for a = 9, b = 4
2  a = a - b; // a = 9 - 4 = 5
3  b = a + b; // b = 5 + 4 = 9
4  a = b - a; // a = 9 - 5
```

We can implement a similar solution with bit manipulation. The benefit of this solution is that it works for more data types than just integers.

```
1  // Example for a = 101 (in binary) and b = 110
2  a = a^b; // a = 101^110 = 011
3  b = a^b; // b = 011^110 = 101
4  a = a^b; // a = 011^101 = 110
```

This code works by using XORs. The easiest way to see how this works is by focusing on a specific bit. If we can correctly swap two bits, then we know the entire operation works correctly.

Let's take two bits, x and y, and walk through this line by line.

1. x = x ^ y

   This line essentially checks if x and y have different values. It will result in 1 if and only if x != y.

2. y = x ^ y

Or: `y = {0 if originally same, 1 if different} ^ {original y}`

Observe that XORing a bit with 1 always flips the bit, whereas XORing with 0 will never change it.

Therefore, if we do `y = 1 ^ {original y}` when `x != y`, then y will be flipped and therefore have x's original value.

Otherwise, if `x == y`, then we do `y = 0 ^ {original y}` and the value of `y` does not change.

Either way, y will be equal to the original value of x.

3. `x = x ^ y`

   Or: `x = {0 if originally same, 1 if different} ^ {original x}`

   At this point, y is equal to the original value of x. This line is essentially equivalent to the line above it, but for different variables.

   If we do `x = 1 ^ {original x}` when the values are different, x will be flipped.

   If we do `x = 0 ^ {original x}` when the values are the same, x will not be changed.

This operation happens for each bit. Since it correctly swaps each bit, it will correctly swap the entire number.

16.2 Word Frequencies: Design a method to find the frequency of occurrences of any given word in a book. What if we were running this algorithm multiple times?

pg 181

SOLUTION

Let's start with the simple case.

Solution: Single Query

In this case, we simply go through the book, word by word, and count the number of times that a word appears. This will take `O(n)` time. We know we can't do better than that since we must look at every word in the book.

```
1  int getFrequency(String[] book, String word) {
2    word = word.trim().toLowerCase();
3    int count = 0;
4    for (String w : book) {
5      if (w.trim().toLowerCase().equals(word)) {
6        count++;
7      }
8    }
9    return count;
10 }
```

We have also converted the string to lowercase and trimmed it. You can discuss with your interviewer if this is necessary (or even desired).

Solution: Repetitive Queries

If we're doing the operation repeatedly, then we can probably afford to take some time and extra memory to do pre-processing on the book. We can create a hash table which maps from a word to its frequency. The frequency of any word can be easily looked up in `O(1)` time. The code for this is below.

```
1  HashMap<String, Integer> setupDictionary(String[] book) {
2    HashMap<String, Integer> table =
```

```
3        new HashMap<String, Integer>();
4     for (String word : book) {
5        word = word.toLowerCase();
6        if (word.trim() != "") {
7           if (!table.containsKey(word)) {
8              table.put(word, 0);
9           }
10          table.put(word, table.get(word) + 1);
11       }
12    }
13    return table;
14 }
15
16 int getFrequency(HashMap<String, Integer> table, String word) {
17    if (table == null || word == null) return -1;
18    word = word.toLowerCase();
19    if (table.containsKey(word)) {
20       return table.get(word);
21    }
22    return 0;
23 }
```

Note that a problem like this is actually relatively easy. Thus, the interviewer is going to be looking heavily at how careful you are. Did you check for error conditions?

16.3 Intersection: Given two straight line segments (represented as a start point and an end point), compute the point of intersection, if any.

pg 181

SOLUTION

We first need to think about what it means for two line segments to intersect.

For two infinite lines to intersect, they only have to have different slopes. If they have the same slope, then they must be the exact same line (same y-intercept). That is:

```
slope 1 != slope 2
OR
slope 1 == slope 2 AND intersect 1 == intersect 2
```

For two straight lines to intersect, the condition above must be true, *plus* the point of intersection must be within the ranges of each line segment.

```
extended infinite segments intersect
AND
intersection is within line segment 1 (x and y coordinates)
AND
intersection is within line segment 2 (x and y coordinates)
```

What if the two segments represent the same infinite line? In this case, we have to ensure that some portion of their segments overlap. If we order the line segments by their x locations (start is before end, point 1 is before point 2), then an intersection occurs only if:

```
Assume:
    start1.x < start2.x && start1.x < end1.x && start2.x < end2.x
Then intersection occurs if:
    start2 is between start1 and end1
```

We can now go ahead and implement this algorithm.

```
Point intersection(Point start1, Point end1, Point start2, Point end2) {
  /* Rearranging these so that, in order of x values: start is before end and
   * point 1 is before point 2. This will make some of the later logic simpler. */
  if (start1.x > end1.x) swap(start1, end1);
  if (start2.x > end2.x) swap(start2, end2);
  if (start1.x > start2.x) {
    swap(start1, start2);
    swap(end1, end2);
  }

  /* Compute lines (including slope and y-intercept). */
  Line line1 = new Line(start1, end1);
  Line line2 = new Line(start2, end2);

  /* If the lines are parallel, they intercept only if they have the same y
   * intercept and start 2 is on line 1. */
  if (line1.slope == line2.slope) {
    if (line1.yintercept == line2.yintercept &&
        isBetween(start1, start2, end1)) {
      return start2;
    }
    return null;
  }

  /* Get intersection coordinate. */
  double x = (line2.yintercept - line1.yintercept) / (line1.slope - line2.slope);
  double y = x * line1.slope + line1.yintercept;
  Point intersection = new Point(x, y);

  /* Check if within line segment range. */
  if (isBetween(start1, intersection, end1) &&
      isBetween(start2, intersection, end2)) {
    return intersection;
  }
  return null;
}

/* Checks if middle is between start and end. */
boolean isBetween(double start, double middle, double end) {
  if (start > end) {
    return end <= middle && middle <= start;
  } else {
    return start <= middle && middle <= end;
  }
}

/* Checks if middle is between start and end. */
boolean isBetween(Point start, Point middle, Point end) {
  return isBetween(start.x, middle.x, end.x) &&
         isBetween(start.y, middle.y, end.y);
}

/* Swap coordinates of point one and two. */
void swap(Point one, Point two) {
  double x = one.x;
  double y = one.y;
```

```
57    one.setLocation(two.x, two.y);
58    two.setLocation(x, y);
59  }
60
61  public class Line {
62    public double slope, yintercept;
63
64    public Line(Point start, Point end) {
65      double deltaY = end.y - start.y;
66      double deltaX = end.x - start.x;
67      slope = deltaY / deltaX; // Will be Infinity (not exception) when deltaX = 0
68      yintercept = end.y - slope * end.x;
69    }
70
71  public class Point {
72    public double x, y;
73    public Point(double x, double y) {
74      this.x = x;
75      this.y = y;
76    }
77
78    public void setLocation(double x, double y) {
79      this.x = x;
80      this.y = y;
81    }
82  }
```

For simplicity and compactness (it really makes the code easier to read), we've chosen to make the variables within `Point` and `Line` `public`. You can discuss with your interviewer the advantages and disadvantages of this choice.

16.4 Tic Tac Win: Design an algorithm to figure out if someone has won a game of tic-tac-toe.

pg 181

SOLUTION

At first glance, this problem seems really straightforward. We're just checking a tic-tac-toe board; how hard could it be? It turns out that the problem is a bit more complex, and there is no single "perfect" answer. The optimal solution depends on your preferences.

There are a few major design decisions to consider:

1. Will `hasWon` be called just once or many times (for instance, as part of a tic-tac-toe website)? If the latter is the case, we may want to add pre-processing time to optimize the runtime of `hasWon`.
2. Do we know the last move that was made?
3. Tic-tac-toe is usually on a 3x3 board. Do we want to design for just that, or do we want to implement it as an `NxN` solution?
4. In general, how much do we prioritize compactness of code versus speed of execution vs. clarity of code? Remember: The most efficient code may not always be the best. Your ability to understand and maintain the code matters, too.

Solution #1: If hasWon is called many times

There are only 3^9, or about 20,000, tic-tac-toe boards (assuming a 3x3 board). Therefore, we can represent our tic-tac-toe board as an `int`, with each digit representing a piece (0 means Empty, 1 means Red, 2 means Blue). We set up a hash table or array in advance with all possible boards as keys and the value indicating who has won. Our function then is simply this:

```
1  Piece hasWon(int board) {
2    return winnerHashtable[board];
3  }
```

To convert a board (represented by a char array) to an `int`, we can use what is essentially a "base 3" representation. Each board is represented as $3^0v_0 + 3^1v_1 + 3^2v_2 + \ldots + 3^8v_8$, where v_i is a `0` if the space is empty, a `1` if it's a "blue spot" and a 2 if it's a "red spot."

```
1  enum Piece { Empty, Red, Blue };
2
3  int convertBoardToInt(Piece[][] board) {
4    int sum = 0;
5    for (int i = 0; i < board.length; i++) {
6      for (int j = 0; j < board[i].length; j++) {
7        /* Each value in enum has an integer associated with it. We
8         * can just use that. */
9        int value = board[i][j].ordinal();
10       sum = sum * 3 + value;
11     }
12   }
13   return sum;
14 }
```

Now looking up the winner of a board is just a matter of looking it up in a hash table.

Of course, if we need to convert a board into this format every time we want to check for a winner, we haven't saved ourselves any time compared with the other solutions. But, if we can store the board this way from the very beginning, then the lookup process will be very efficient.

Solution #2: If we know the last move

If we know the very last move that was made (and we've been checking for a winner up until now), then we only need to check the row, column, and diagonal that overlaps with this position.

```
1  Piece hasWon(Piece[][] board, int row, int column) {
2    if (board.length != board[0].length) return Piece.Empty;
3
4    Piece piece = board[row][column];
5
6    if (piece == Piece.Empty) return Piece.Empty;
7
8    if (hasWonRow(board, row) || hasWonColumn(board, column)) {
9      return piece;
10   }
11
12   if (row == column && hasWonDiagonal(board, 1)) {
13     return piece;
14   }
15
16   if (row == (board.length - column - 1) && hasWonDiagonal(board, -1)) {
17     return piece;
18   }
```

```
19
20    return Piece.Empty;
21  }
22
23  boolean hasWonRow(Piece[][] board, int row) {
24    for (int c = 1; c < board[row].length; c++) {
25      if (board[row][c] != board[row][0]) {
26        return false;
27      }
28    }
29    return true;
30  }
31
32  boolean hasWonColumn(Piece[][] board, int column) {
33    for (int r = 1; r < board.length; r++) {
34      if (board[r][column] != board[0][column]) {
35        return false;
36      }
37    }
38    return true;
39  }
40
41  boolean hasWonDiagonal(Piece[][] board, int direction) {
42    int row = 0;
43    int column = direction == 1 ? 0 : board.length - 1;
44    Piece first = board[0][column];
45    for (int i = 0; i < board.length; i++) {
46      if (board[row][column] != first) {
47        return false;
48      }
49      row += 1;
50      column += direction;
51    }
52    return true;
53  }
```

There is actually a way to clean up this code to remove some of the duplicated code. We'll see this approach in a later function.

Solution #3: Designing for just a 3x3 board

If we really only want to implement a solution for a 3x3 board, the code is relatively short and simple. The only complex part is trying to be clean and organized, without writing too much duplicated code.

The code below checks each row, column, and diagonal to see if there is a winner.

```
1   Piece hasWon(Piece[][] board) {
2     for (int i = 0; i < board.length; i++) {
3       /* Check Rows */
4       if (hasWinner(board[i][0], board[i][1], board[i][2])) {
5         return board[i][0];
6       }
7
8       /* Check Columns */
9       if (hasWinner(board[0][i], board[1][i], board[2][i])) {
10        return board[0][i];
11      }
```

```
    }

    /* Check Diagonal */
    if (hasWinner(board[0][0], board[1][1], board[2][2])) {
      return board[0][0];
    }

    if (hasWinner(board[0][2], board[1][1], board[2][0])) {
      return board[0][2];
    }

    return Piece.Empty;
  }

  boolean hasWinner(Piece p1, Piece p2, Piece p3) {
    if (p1 == Piece.Empty) {
      return false;
    }
    return p1 == p2 && p2 == p3;
  }
```

This is an okay solution in that it's relatively easy to understand what is going on. The problem is that the values are hard coded. It's easy to accidentally type the wrong indices.

Additionally, it won't be easy to scale this to an NxN board.

Solution #4: Designing for an NxN board

There are a number of ways to implement this on an NxN board.

Nested For-Loops

The most obvious way is through a series of nested for-loops.

```
Piece hasWon(Piece[][] board) {
  int size = board.length;
  if (board[0].length != size) return Piece.Empty;
  Piece first;

  /* Check rows. */
  for (int i = 0; i < size; i++) {
    first = board[i][0];
    if (first == Piece.Empty) continue;
    for (int j = 1; j < size; j++) {
      if (board[i][j] != first) {
        break;
      } else if (j == size - 1) { // Last element
        return first;
      }
    }
  }

  /* Check columns. */
  for (int i = 0; i < size; i++) {
    first = board[0][i];
    if (first == Piece.Empty) continue;
    for (int j = 1; j < size; j++) {
      if (board[j][i] != first) {
```