1 Fun with Hash Functions (from 61BL SU2014 MT2)

Here are three potential implementations of the `Integer`'s `hashCode()` function. Categorize each as either a valid or an invalid hash function. If it is invalid, explain why. If it is valid, point out a flaw/disadvantage.

Note: A "valid" `hashCode()` means that: any two Integers that are `.equals()` to each other should also return the same hash code value.

Another note: the `Integer` class extends the `Number` class, a direct subclass of `Object`. The `Number` class' hashCode() method directly calls the `Object` class' hashCode() method.

(a) `public int hashCode() {return -1; }`

```
Valid. As required, this hash function returns the same hashCode for
    Integers that are .equals() to each other. However, this is a
    terrible hash code because collisions are extremely frequent
    (collisions occur 100\% of the time).
```

(b) `public int hashCode() {return intValue() * intValue(); }`

```
Valid. Similar to (a), this hash function returns the same hashCode
    for Integers that are .equals(). However, Integers that share the
    same absolute values will collide (for example, x=5 and x=-5 will
    have the same hash code). A better hash function would be to just
    return the intValue() itself.
```

(c) `public int hashCode() {return super.hashCode(); }`

```
Invalid. This is not a valid hash function because Integers that are
    .equals() to each other will not have the same hash code. Instead,
    this hash function returns some integer corresponding to the
    Integer object's location in memory.
```

2 HashMap Modification (from 61BL SU2010 MT2)

(a) When you modify a key that has been inserted into a HashMap will you be able to retrieve that entry again? Explain.

☐ Always [✓ Sometimes] ☐ Never

(b) When you modify a value that has been inserted into a HashMap will you be able to retrieve that entry again? Explain.

[✓ Always] ☐ Sometimes ☐ Never

3 Analyzing Proj1 ArrayDeque's Runtime

Recall the `ArrayDeque` from proj1. Our implementation uses a circular array with two pointers denoting the front and back of the `ArrayDeque`. Starting with an initial size of 8, the array doubles in size when it reaches full capacity, and halves in size when it's load factor is lower than 0.25. Resizing will reposition the elements to start from index 0 for ease of maintenance. Fill in this table with best, worst, and average case runtimes of the `ArrayDeque` methods (in Θ notation).

	Best Case	Worst Case	Average Case
addFirst/Last	$\Theta(1)$	$\Theta(N)$	$\Theta(1)$
rmFirst/Last	$\Theta(1)$	$\Theta(N)$	$\Theta(1)$
get	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

4 Recursion and Dynamic Programming

Implement Fibonacci using memoization (memorizing previously solved problems).

```
public class Fibonacci {
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

    // For your reference, fib(0) = 0, fib(1) = 1, fib(2) = 1, ...
    public int fib(int n) {
        if (n == 0) {
            return 0;
        } else if (n == 1) {
            return 1;
        } else if (map.containsKey(n)) {
            return map.get(n);
        }

        int result = fib(n - 2) + fib(n - 1);
        map.put(n, result);
        return result;
    }
}
```

5 Hashing a Tic-Tac-Toe Board (Bonus)

Given the provided (minimal) implementations below, write the `.hashCode()` and `.equals()` methods for the `Piece` and `Board` classes (we did `Piece.equals()` for you). Try to ensure that different board configurations have different hash codes.

```java
public class Piece {
    private String type; // Will be either "X" or "O".

    /** Some other code... */

    public boolean equals(Object o) {
        Piece otherPiece = (Piece) o;
        return this.type.equals(otherPiece.type);
    }

    public int hashCode() {
        if (type.equals("X")) {
            return 2;
        } else if (type.equals("O")) {
            return 1;
        }
        return 0;
    }
}

public class Board {
    public static final int SIZE = 3; // Tic Tac Toe Boards are always 3x3

    private Piece[][] pieces;
    private boolean isXTurn;

    /** Some other code... */

    public int hashCode() {
        int code = 0;
        if (isXTurn) {
            code = 1;
        }

        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                Piece currentPiece = pieces[i][j];
                code *= 3;
                if (currentPiece != null) {
                    code += currentPiece.hashCode();
                }
            }
        }

        return code;
    }
}
```

Bite-sized Bonus: How do you implement the `.equals()` method for `Board`?

`Take a [`**`double`**`]` **`for`** `loop through the pieces array and ensure that at every spot, either they are both nulls, both X's, or both O's. We didn't leave space in the above code because we tried to keep everything on one page...`

Bigger Bonus: Is it possible to perform a "perfect hash"? If we now wanted to have three different types of pieces, X's, O's and Triangles, does that change your answer?

`Yes, there are a combined` 2×3^9 `possible configurations, which can be uniquely represented using the hash code. To include a` **`new`** `type of piece, use powers of 4 to represent the possible pieces in each board location.`