1 Reduce

We’d like to write a method `reduce`, which uses a `BinaryFunction` interface to accumulate the values of a `List` of integers into a single value. `BinaryFunction` can operate (through the `apply` method) on two integer arguments and return a single integer. Note that `reduce` can now work with a range of binary functions (for example, addition and multiplication). Write two classes `Adder` and `Multiplier` that implement `BinaryFunction`. Then, fill in `reduce` and `main`, and define types for `add` and `mult` in the space provided.

```java
import java.util.ArrayList;
import java.util.List;
public class ListUtils {
    /** If the list is empty, return 0.
     * If it has one element, return that element.
     * Otherwise, apply a function of two arguments cumulatively to the
     * elements of list and return a single accumulated value.
     * Does not modify the list. */
    public static int reduce(BinaryFunction func, List<Integer> list) {
        // Implementation goes here
    }

    public static void main(String[] args) {
        ArrayList<Integer> integers = new ArrayList<>();
        integers.add(2); integers.add(3); integers.add(4);
        __________ add = ____________________;
        __________ mult = ____________________;
        reduce(add, integers); // Should evaluate to 9
        reduce(mult, integers); // Should evaluate to 24
    }
}

interface BinaryFunction {
    int apply(int x, int y);
}
```
// Add additional classes below:
2 Inheritance Infiltration

Access modifiers are critical when it comes to security. Look at the PasswordChecker and User classes below.

```java
public class PasswordChecker {
    /** Returns true if the provided login and password are correct. */
    public boolean authenticate(String login, String password) {
        // Does some secret authentication stuff...
    }
}

public class User {
    private String username;
    private String password;
    public void login(PasswordChecker p) {
        p.authenticate(username, password);
    }
}
```

Even though the username and password variables are private, the login and authenticate methods are both public. We can use inheritance to take advantage of this and extract the password of any given User object. Complete the PasswordExtractor class below so that calling extractPassword returns the password of a given User. You may not modify the provided classes (i.e. you may not change the implementations of PasswordChecker or User).

```java
public class PasswordExtractor extends PasswordChecker {
    String extractedPassword;
    public String extractPassword(User u) {
        // Are there any other methods that we need to implement?
    }
}
```

**Hint:** The login method of User passes in the username and password fields as parameters to the authenticate method of a given PasswordChecker. Think about how we can take advantage of method overriding to gain access to the password.
3 A Bit on Bits

Recall the following bit operations and shifts:

1. Mask \((x \& y)\): yields 1 only if both bits are 1.
   \[01110 \& 10110 = 00110\]

2. Set \((x \mid y)\): yields 1 if at least one of the bits is 1.
   \[01110 \mid 10110 = 11110\]

3. Flip \((x \mathbin{\hat{\lor}} y)\): yields 1 only if the bits are different.
   \[01110 \mathbin{\hat{\lor}} 10110 = 11000\]

4. Flip all \((\mathbin{\hat{\neg}} x)\): turns all 1’s to 0 and all 0’s to 1.
   \[\mathbin{\hat{\neg}} 01110 = 10001\]

5. Left shift \((x << \text{left}_\text{shift})\): shifts the bits to the left by \text{left}_\text{shift} places, filling in the right with zeros.
   \[10110111 << 3 = 10111000\]

6. Arithmetic right shift \((x >> \text{right}_\text{shift})\): shifts the bits to the right by \text{right}_\text{shift} places, filling in the left bits with the current existing leftmost bit.
   \[10110111 >> 3 = 11110110\]
   \[00110111 >> 3 = 00000110\]

7. Logical right shift \((x >>> \text{right}_\text{shift})\): shifts the bits to the right by \text{right}_\text{shift} places, filling in the left with zeros.
   \[10110111 >>> 3 = 00010110\]
Implement the following two methods. For both problems, i=0 represents the least significant bit, i=1 represents the bit to the left of that, and so on.

(a) Implement `isBitOn` so that it returns a boolean indicating if the ith bit of `num` has a value of 1. For example, `isBitOn(2, 0)` should return false (the 0th bit is 0), but `isBitOn(2, 1)` should return true (the 1st bit is 1).

```java
/** Returns whether the ith bit of num is a 1 or not. */
public static boolean isBitOn(int num, int i) {
    int mask = 1 ________________________________;
    return ________________________________;
}
```

(b) Implement `turnBitOn` so that it returns the input number but with its ith significant bit set to a value of 1. For example, if `num` is 1 (1 in binary is 01), then calling `turnBitOn(1, 1)` should return the binary number 11 (aka 3).

```java
/** Returns the input number but with its ith bit changed to a 1. */
public static int turnBitOn(int num, int i) {
    int mask = 1 ________________________________;
    return ________________________________;
}
```
4 Flip Halves Extra

Given two numbers, a and b, in bit representation, return a new number whose first half is the first half of a flipped, and second half is the second half of b flipped. As usual, assume all numbers are 4 bytes, or 32 bits. See below for an example.

```java
int a = 0x88888888; // Looks like 0b1000_1000_1000 ...
int b = 0x33333333; // Looks like 0b0011_0011_0011 ...
int expected_answer = 0x7777CCCC; // Looks like 0b0111_0111_ ... _1100_1100

Implement flip_halves below

public int flip_halves(int a, int b) {

}
```