CS 61B Spring 2022

Packages & Bits

Discussion 6: February 21, 2022

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.

```
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) {









    }
    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:
```

```
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) {
        if (list.size() == 0) { return 0; }
        int soFar = list.get(0);
        for (int i = 1; i < list.size(); i++) {
            soFar = func.apply(soFar, list.get(i));
        }
        return soFar;
    }
    public static void main(String[] args) {
        ArrayList<Integer> integers = new ArrayList<>();
        integers.add(2); integers.add(3); integers.add(4);
        Adder add = new Adder();
        Multiplier mult = new Multiplier();
        reduce(add, integers); //Should evaluate to 9
        reduce(mult, integers); //Should evaluate to 24
    }
}

interface BinaryFunction {
    int apply(int x, int y);
}
public class Adder implements BinaryFunction {
    public int apply(int x, int y) {
        return x + y;
    }
}
public class Multiplier implements BinaryFunction {
    public int apply(int x, int y) {
        return x * y;
    }
}
```

We declare an interface `BinaryFunction` which our `Adder` and `Multiplier` classes can implement. Writing a common interface is important, because it allows us to write a `reduce` function that is capable of accepting many kinds of functions. Note that interface methods are `public` by default, so `apply` must be `public` in `Adder` and `Multiplier`.

2 Inheritance Infiltration

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

```
1  public class PasswordChecker {
2      /** Returns true if the provided login and password are correct. */
3      public boolean authenticate(String login, String password) {
4          // Does some secret authentication stuff...
5      }
6  }
7
8  public class User {
9      private String username;
10     private String password;
11     public void login(PasswordChecker p) {
12         p.authenticate(username, password);
13     }
14 }
```

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`).

```
public class PasswordExtractor extends ____________________ {
   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.

```
1
2  public class PasswordExtractor extends PasswordChecker {
3      String extractedPassword;
4
5      public String extractPassword(User u) {
6          u.login(this);
7          return extractedPassword;
8      }
9
10     @Override
11     public boolean authenticate(String login, String password) {
12         extractedPassword = password; // Victory is mine >:)
13         return true; // or false. Needs to return something to compile.
14     }
15
16 }
```

By letting us subclass `PasswordChecker`, we can overwrite the `authenticate` method to capture the password in a local variable. By calling a user's `login` method and passing ourselves in, we can force the user to provide its password. Finally, we can return the extracted password. We could fix this security hole by making `PasswordChecker` no longer a public class.

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 | y`): yields 1 if at least one of the bits is 1.
   `01110 | 10110 = 11110`
3. Flip (`x ^ y`): yields 1 only if the bits are different.
   `01110 ^ 10110 = 11000`
4. Flip all (`~ x`): turns all 1's to 0 and all 0's to 1.
   `~ 01110 = 10001`
5. Left shift (`x << left_shift`): shifts the bits to the left by `left_shift` places, filling in the right with zeros.
   `10110111 << 3 = 10111000`
6. Arithmetic right shift (`x >> right_shift`): shifts the bits to the right by `right_shift` places, filling in the left bits with the current existing leftmost bit.
   `10110111 >> 3 = 11110110`
   `00110111 >> 3 = 00000110`
7. Logical right shift (`x >>> right_shift`): shifts the bits to the right by `right_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 `isBitIOn` so that it returns a boolean indicating if the `ith` bit of `num` has a value of 1. For example, `isBitIOn(2, 0)` should return `false` (the 0th bit is 0), but `isBitIOn(2, 1)` should return `true` (the 1st bit is 1).

```
/** Returns whether the ith bit of num is a 1 or not. */
public static boolean isBitIOn(int num, int i) {

    int mask = 1 ____________________________________________;

    return ___________________________________________________;
}
```

```
1  /** Returns whether the ith bit of num is a 1 or not. */
2  public static boolean isBitIOn(int num, int i) {
3      int mask = 1 << i;
4      return (mask & num) != 0;
5  }
```

(b) Implement `turnBitIOn` 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 `turnBitIOn(1, 1)` should return the binary number 11 (aka 3).

```
/** Returns the input number but with its ith bit changed to a 1. */
public static int turnBitIOn(int num, int i) {

    int mask = 1 ____________________________________________;

    return ___________________________________________________;
}
```

```
1  /** Returns the input number but with its ith bit changed to a 1. */
2  public static int turnBitIOn(int num, int i) {
3      int mask = 1 << i;
4      return num | mask;
5  }
```

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.

```
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) {



}
```

Solution:

```
public int flip_halves(int a, int b) {
   int mask = 0xFFFF0000;
   return ((mask ^ a) & mask) | ((~mask ^ b) & ~mask);
}
```

Note that you can switch the ordering of the ^ and & as well.