CS61B, Fall 2014 Test #1 Solution P. N. Hilfinger and Josh Hug

Reference Material.

```
public class IntList {
    /** First element of list. */
    public int head;
    /** Remaining elements of list. */
    public IntList tail;

    /** A List with head HEAD0 and tail TAIL0. */
    public IntList(int head0, IntList tail0)
    { head = head0; tail = tail0; }

    /** A List with null tail, and head = 0. */
    public IntList() { this (0, null); }

    /** Returns a new IntList containing the ints in ARGS. */
    public static IntList list(Integer ... args) {
           // Implementation not shown
    }

    /** Non-destructively returns a new IntList containing all my
     *  values that have indices >= START and < END. Undefined if
     *  any of the required items are non-existent or I is null.
     *  The result shares no objects with me. */
     public static IntList sublist(IntList L, int start, int end) {
           // Implementation not shown
    }

    @Override
    /** Returns true iff L is a list with the same items as
     *  this list (as determined by their .equals methods) in the
     *  same order. */
    public boolean equals(Object L) {
           // Implementation not shown
    }

}
```

1. [2 points] Assume that a `Point`'s `toString` method returns a string containing that `Points`'s coordinates (so that `System.out.println(x)` prints "`(4, 5)`" if `x` is `new Point(4, 5)` and "`null`" if `x` is null). What is the output of the following (valid) program?

```
import java.awt.Point;
public class Foo {
    public static void bar (Point[] arr, Point p) {
        arr[1] = p;
        arr[2] = arr[1];
        p.x = 1;
        p = new Point(2,2);
        p.y = 3;
        arr[3] = p;
    }
    public static void main(String[] args){
        Point[] points = new Point[4];
        Point p = new Point(0,0);
        bar(points, p);
        System.out.println(p);
        for (int i = 0; i < points.length; i += 1) {
            System.out.println(points[i]);
        }
    }
}
```

Answer:

`(1, 0)`

`null`

`(1, 0)`

`(1, 0)`

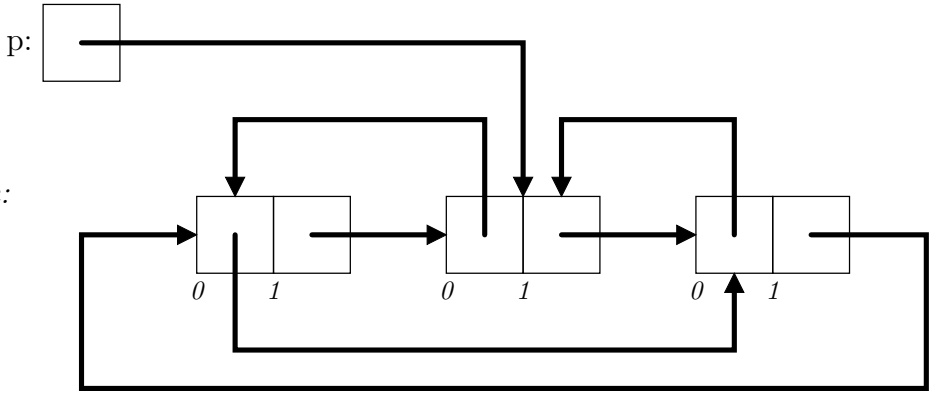
`(2, 3)`

2. [2 points] For each of the following, fill in the blanks on the right to convert the "Before" diagram into the "After" diagram. **Do not introduce any new variables other than those shown in the diagrams**. Put at most one statement or expression in each blank. You need not use all the blanks.

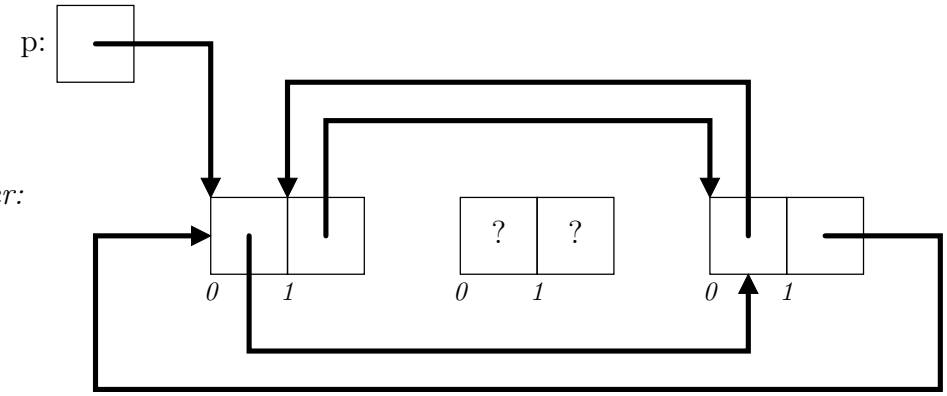
Notation: (1) A '?' indicates that a value is allowed to be anything you want. (2) Arrows point to whole objects, not individual fields.

a. In the following, `p` is declared "`Object[] p;`." That is, the objects are all *arrays*, not `IntLists`. *WARNING:* since the elements of these array objects have static type `Object`, you may have to cast them before you can perform other operations on them.

Before:



After:

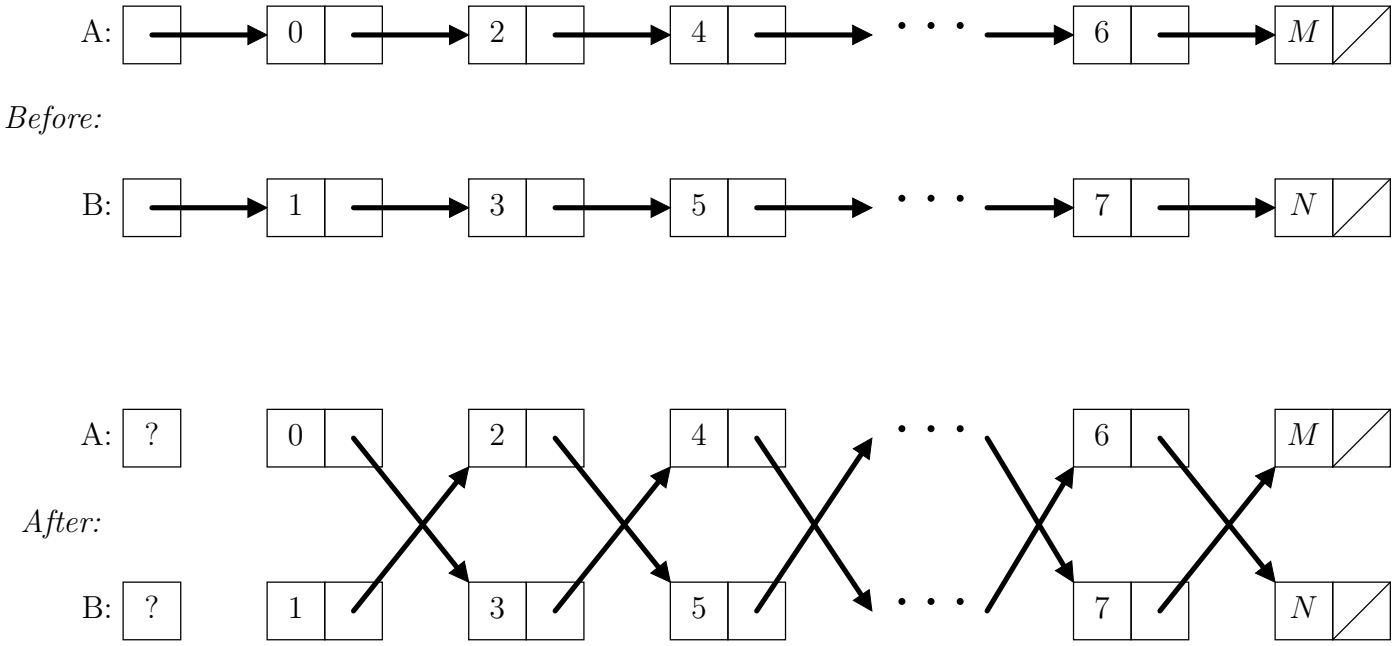


`((Object[]) p[0])[1] = p[1];` _____

`((Object[]) p[1])[0] = p[0];` _____

`p = (Object[]) p[0];` _____

b. Here, `A` and `B` are declared to be `IntLists`, and the two boxes in each object are respectively the `head` and `tail` of an `IntList`. Do not change any of the values in the `head` fields. Do not create new objects. Assume that all four rows have the same number of objects (so that $N = M + 1$).




```
while (A != null)                                    {

    IntList t1 = A.tail;

    A.tail = B.tail;

    B.tail = t1;

    B = A.tail;

    A = t1;
}
```

3. [2 points] When the main program of `Foo` runs, it is supposed to print out the following lines:

```
1
2
3
Superdog
Superdog
bark 3
4
```

However, the `Animal` and `Dog` classes are incorrect. Some parts—the blanks—are missing, and you must fill these in. Other lines are incorrect (don't compile or don't produce the right output). Cross these out and rewrite each with a single (corrected) line of code.

```
/***  Class Foo  ***/

import zoo.Animal;
import housepets.Dog;

public class Foo { // Do not modify class Foo

    public static void main(String[] args) {
        Animal a = new Dog();
        Animal b = new Dog();
        Animal c = new Dog();
        a.makeNoise();
        b.makeNoise();
        a.makeNoise();
        c.sayName();
        a.sayName();
        a.makeNoise("bark");
        c.makeNoise();
    }
}
```

Classes `Animal` *and* `Dog` *are on the next page. Again, fill in the blanks and cross out and rewrite lines of code to make them work properly.*

```
package zoo;

public class Animal {
    protected static int noise;  // CHANGED FROM int noise;
    private String name; // Do not modify this line

    public Animal(String name) {
        this.name = name;  // CHANGED FROM name = name;
    }

    public void makeNoise() {

        noise += 1;             ;

        System.out.println(noise                    );
    }

    public void sayName() {
        System.out.println(name);
    }


    public void makeNoise(String sound){}
}
```

```
package housepets;
import zoo.Animal;

public class Dog extends Animal {  // CHANGED FROM public class Dog {
    public Dog() {

        super("Superdog");                  ;
    }

    public void makeNoise(String sound) {

        System.out.println(sound + " " + noise                    );
    }
}
```

4. [1 points] Bob isn't sure how to test that `sublist` is nondestructive, but the Javadocs for `sublist` are pretty adamant about making sure it is (see the Reference Material on page 1 of this test). Help him out by filling in the two blank lines to ensure that `IntList.sublist` really is non-destructive. In addition, you may cross out and rewrite up to one line of the existing code (if you find it in error, that is).

```
@Test
public void testSublist() {
    IntList x012345 = IntList.list(0, 1, 2, 3, 4, 5);
    IntList copyOfx012345 = x012345;   // WRONG
    // REPLACE WITH:
    IntList copyOfx012345 = IntList.list(0, 1, 2, 3, 4, 5);

    assertEquals(copyOfx012345, IntList.sublist(x012345, 0, 6));


    assertEquals(x012345, copyOfx012345);

    IntList x45 = IntList.list(4, 5);
    assertEquals(x45, IntList.sublist(x012345, 4, 6));


    assertEquals(0x12345, copyOfx012345);
}
```

5. [2 points] Fill in the `next()` method in the following class. Do not modify anything outside of `next`. [Language note: Java automatically converts between `int` and `Integer`.]

```
import java.util.Iterator;
import java.util.NoSuchElementException;
/** Iterates over every Kth element of the IntList given to the constructor.
 *  For example, if L is an IntList containing elements
 *  [0, 1, 2, 3, 4, 5, 6, 7] with K = 2, then
 *      for (Iterator<Integer> p = new KthIntList(L, 2); p.hasNext(); ) {
 *          System.out.println(p.next());
 *      }
 *  would print get 0, 2, 4, 6.  */
public class KthIntList implements Iterator<Integer> {
    public int k;
    private IntList curList;
    private boolean hasNext;

    public KthIntList(IntList I, int k) {
        this.k = k; this.curList = I; this.hasNext = true;
    }

    /** Returns true iff there is a next Kth element. Do not modify. */
    public boolean hasNext() {
        return this.hasNext;
    }

    /** Returns the next Kth element of the IntList given in the constructor.
     *  Returns the 0th element first. Throws a NoSuchElementException if
     *  there are no Integers available to return. */
    public Integer next() {
        if (curList == null) {
            throw new NoSuchElementException();
        }
        Integer toReturn = curList.head;
        for (int i = 0; i < k && curList != null; i++) {
            curList = curList.tail;
        }
        hasNext = (curList != null);
        return toReturn;
    }
}
```

6. [2 point] The `bitwiseMultiply` method takes in two 32-bit integers and returns their product. The process is just like ordinary multiplication of decimals. For example, restricting ourselves to 8 bits for simplicity, we could compute 12×9 (in binary. 1100×1001 like this:

```
  00001100
× 00001001
----------
  00001100
+ 01100
----------
  01101100
```

Fill in the blanks so that the method works. You may only use the operators

`==` `!=` `&` `|` `^` `<<` `>>` `>>>`

in your solution.

```
public int bitwiseMultiply(int a, int b) {
    int result;
    result = 0;
    while (b != 0) {

        if ((b & 1) == 1                          ) {
            result = result + a;
        }

        a = a << 1;                                          ;

        b = b >>> 1;                                         ;
    }

    return result;
}
```

7. [1 point] In the Ada programming language, integer decimal numerals may (but need not) contain underscores (_), which are ignored, but serve to make the numeral more readable (like commas in ordinary decimal numbers). For example, `1_023_800` is such a numeral. The underscores are restricted in two ways:

1. They may not appear at the beginning or end of a numeral. For example, `120_` and `_120` are invalid.

2. Underscores may not be adjacent. For example, `1__300` is invalid.

Write a Java regular expression that matches all and only non-negative integer decimal numerals in this format. It is not necessary to have three digits between underscores. Your numerals should have at least one digit.

```
Answer: \d(_?\d)*   or, as a string literal: "\\d(_?\\d)*"
```

8. [1 point] What do the following familiar phrases have in common?

"break the ice"
"eaten me out of house and home"
"elbow room"
"cold comfort"
"dead as a doornail"

Answer: They are all credited to Shakespeare. (Yes, they are idioms, figures of speech, etc., but not just *any* idioms, figures of speech, etc.)

9. [3 points] For part (a) and (b) below, give an asymptotic $\Theta(\cdot)$ bound for the number of calls to `println` as a function of the argument `n`.

a.
```
public static void printIndices(int n) {
    for (int i = 0; i < n; i += 1) {
        for (int j = 0; j < n*n; j += 1) {
            System.out.println("This is " + i + ", " + j);
        }
    }
}
```

Bound: $\Theta(n^3)$

b.
```
public static void printIndices2(int n) {
    for (int i = n; i > 1; i = i/2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("Printing something");
            int A = 1 + 1;
        }
    }
}
```

Bound: $\Theta(N)$

Problem continues on the next page.

For c–e, give $\Theta(\cdot)$ bounds on the execution time of `wordCount` as a function of N, the length of the parameter `words`. Assume that all of the Strings in `words` have lengths bounded by a constant. That is, for some constant K, `words`$[i]$.`length`$() \leq K$ for all i.

c.
```
import java.util.Arrays;

public class Asymptotics {
    public static int wordCount(String[] words) {
        /* Assume that Arrays.sort is Θ(N lg N) */
        Arrays.sort(words);
        int N = words.length;
        int wordCount = 0;
        int i = 0;
        while (i < N) {
            String thisWord = words[i];
            wordCount += 1;
            int j = i + 1;
            while (j < N) {
                if (!words[j].equals(thisWord))
                    break;
                j++;
            }
            i = j;
        }
        return wordCount;
    }
}
```

Bound: $\Theta(N \lg N)$

d. Suppose that we replace `Arrays.sort` with another sorting method that is $\Theta(N)$ Does this affect the $\Theta(\cdot)$ runtime of `wordCount`? If so, what is the new runtime bound? Explain your answer. **Answer:** Yes; the bound becomes $\Theta(N)$. The nested while loop looks at each word only once, and so is $\Theta(N)$.

e. Suppose we replace `Arrays.sort` with a $\Theta(N^3)$ sorting method. Does this affect the $\Theta(\cdot)$ runtime of `wordCount`? If so, what is the new runtime bound? Explain your answer. **Answer:** Yes; the bound becomes $\Theta(N^3)$. This time, the cost of sorting dominates the cost of finding unique words.

10. [2 points] Consider the following interface:

```
public interface IteratedFunction {
    int apply(int x);
    /** The result of apply(apply(...apply(X))), where apply is called
     *  N times.  Assumes N >= 0. */
    int applyN(int x, int n);
}
```

a. Create an abstract class `AbstractIteratedFunction` that implements the interface and provides a default implementation of one of these methods in terms of the other (your choice), leaving the other method abstract.

```
// Solution:
public abstract class AbstractIteratedFunction implements IteratedFunction {
    public int applyN(int x, int n) {
        while (n > 0) {
            x = apply(x);
            n -= 1;
        }
        return x;
    }
}
```

b. Now create a concrete implementation of `AbstractIteratedFunction` called `Collatz`. `Collatz`'s one-argument `apply` should return $x/2$ if x is even, or $3x + 1$ if x is odd. Therefore, `applyN` on a `Collatz` object should apply the `Collatz` function n times.

```
// Solution:
public class Collatz extends AbstractIteratedFunction {
    public int apply(int x) {
        return x % 2 == 0 ? x / 2 : 3 * x + 1;
        // or: if (x % 2) == 0 {
        //         return x / 2;
        //     else {
        //         return 3 * x + 1;
        //     }
    }
}
```