CSE 143, Summer 2012
Final Exam - Part 1
Thursday, August 16, 2012

Personal Information:

Name: _____

Section: _____ **TA:** _____

Student ID #: _____

- You have 60 minutes to complete this exam.
  You may receive a deduction if you keep working after the instructor calls for papers.
- This exam is open-book for the *Building Java Programs* textbook, but otherwise it is closed-book/notes.
  You may not use any paper resources or any computing devices including calculators.
- Code will be graded on proper behavior/output and not on style, unless otherwise indicated.
- Do not abbreviate code, such as "ditto" marks or dot-dot-dot ... marks.

  The only abbreviations that *are* allowed for this exam are:
  - `S.o.p` for `System.out.print`, and
  - `S.o.pln` for `System.out.println`.
- You do not need to write `import` statements in your code.
- If you enter the room, you must turn in an exam before leaving the room.
- You must show your Student ID to a TA or instructor for your exam to be accepted.

Good luck!

Score summary: (for grader only)

Problem	Description	Earned	Max
1	Inheritance/Polymorphism Mystery		15
2	Comparable Programming		15
3	Binary Search Trees		7
4	Binary Tree Programming		15
X	Extra Credit		+1
TOTAL	**Total Points**		**100**

1. Inheritance and Polymorphism

Consider the following classes:

```
public class Polar extends Bear {
    public void b() {
        a();
        System.out.println("Polar 2");
    }

    public void c() {
        System.out.println("Polar 3");
    }
}

public class Bear {
    public void a() {
        System.out.println("Bear 1");
        c();
    }

    public void c() {
        System.out.println("Bear 3");
    }
}

public class Pooh extends Teddy {
    public void b() {
        System.out.println("Pooh 2");
        super.c();
    }

    public void c() {
        System.out.println("Pooh 3");
    }
}

public class Teddy extends Bear {
    public void a() {
        System.out.println("Teddy 3");
        super.a();
    }
}
```

and that the following variables are defined:

```
Bear   var1 = new Teddy();
Pooh   var2 = new Pooh();
Bear   var3 = new Polar();
Bear   var4 = new Pooh();
Object var5 = new Teddy();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "a / b / c" to indicate three lines of output with "a" followed by "b" followed by "c". If the statement causes an error, fill in the right-hand column with **"error"** to indicate this.

Statement	**Output**
`var1.a();`	_____
`var1.c();`	_____
`var2.a();`	_____
`var2.b();`	_____
`var3.a();`	_____
`var3.b();`	_____
`var4.a();`	_____
`var5.a();`	_____
`((Pooh) var5).a();`	_____
`((Teddy) var1).a();`	_____
`((Teddy) var4).a();`	_____
`((Pooh) var3).b();`	_____
`((Polar) var3).b();`	_____
`((Teddy) var4).c();`	_____
`((Bear) var5).c();`	_____

2. `Comparable` Programming

Define a class called **`Office`** that keeps track of information about an office's features. The class should have the following public methods:

Methods	Description
`public` **`Office`**`(double width, double length, boolean couch, int windows)`	constructs an `Office` object with the given width and length, couch (true or false) and number of windows
`public boolean` **`isCorner`**`()`	returns `true` if the office is a corner office. A corner office is square (has the same width and length) and has exactly two windows.
`public String` **`toString`**`()`	returns a string representing the office in the format: width: <width>, length: <length>, windows: <windows>. If the office has a couch ", has a couch" should be appended to the end of the string.

Examples:

```
Office o1 = new Office (10.3, 10.3, true, 3);
```

A call on `o1.toString()` would return:

```
width: 10.3, length: 10.3, windows: 3, has a couch
```

```
Office o2 = new Office (14.0, 6.7, false, 3);
```

A call on `o2.toString()` would return:

```
width: 14.0, length: 6.7, windows: 2
```

Also **make `Office` objects comparable to each other using the `Comparable` interface**. `Offices` that have that have a greater area (width * length) should be considered "less" than other `Offices` so that they appear at the beginning of a sorted list. `Offices` that have the same area should be ordered by the number of windows they have, with `Offices` that have a greater amount of windows considered "less" than `Offices` that have less. If `Offices` still appear equal they should be compared by whether or not they have a couch. `Offices` with couches should be considered "less".

(Write your answer on the next page.)

---

2. `Comparable` Programming (writing space)

3. Binary Search Trees

(a) Write the binary search tree that would result if these elements were added to an empty tree in this order:

- Glinda, Cowardly lion, Dorothy, Toto, Scarecrow, Tin man, Wicked witch, Wizard, Boq, Ozma

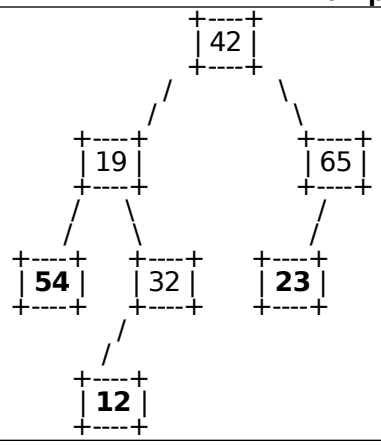
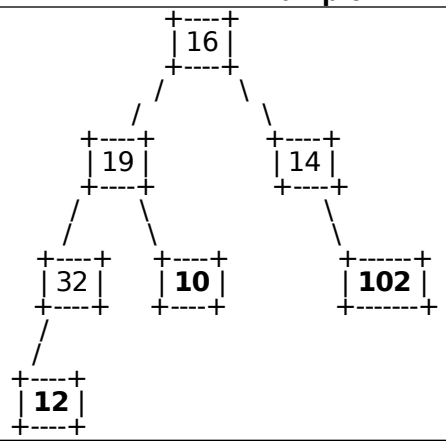
(b) Write the elements of your above tree in the order they would be visited by each kind of traversal:

- Pre-order: _____
- In-order: _____
- Post-order: _____

4. Binary Tree Programming

Write a method **`sumLeaves`** to be added to the `IntTree` class from class (see cheat sheet). Your method should return the sum of the data stored in all of the leaf nodes. Your method should not alter the tree.

```
IntTree tree = new IntTree();
...
tree.sumLeaves();
```

Example 1	Example 2
	
Return: 89	Return: 124

If the tree is empty a call to your method should return 0;

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the tree class nor create any data structures such as arrays, lists, etc. You should not construct any new node objects or change the `data` of any nodes. For full credit, your solution must be recursive.

(Write your answer on the next page.)

4. Binary Tree Programming (writing space)

X. Extra Credit

Make a guess what grade you will get on this half of the test. Guess what the average will be on this test.

(This is just for fun. You will get credit for guessing no matter how correct or incorrect your guess is.)

My Grade:_____

Class Average:_____

^_^ CSE 143 FINAL EXAM CHEAT SHEET ^_^

Constructing Various Collections

```
List<Integer> list = new ArrayList<Integer>();
Queue<Double> queue = new LinkedList<Double>();
Stack<String> stack = new Stack<String>();
Set<String> set = new HashSet<String>();
Map<String, Integer> map = new TreeMap<String, Integer>();
```

Methods Found in ALL collections (Lists, Stacks, Queues, Sets, Maps)

`clear()`	removes all elements of the collection
`equals(`**`collection`**`)`	returns `true` if the given other collection contains the same elements
`isEmpty()`	returns `true` if the collection has no elements
`size()`	returns the number of elements in the collection
`toArray()`	returns an array of the elements in this collection
`toString()`	returns a string representation such as `"[10, -2, 43]"`

Methods Found in both Lists and Sets (ArrayList, LinkedList, HashSet, TreeSet)

`add(`**`value`**`)`	adds value to collection (appends at end of list)
`contains(`**`value`**`)`	returns `true` if the given value is found somewhere in this collection
`iterator()`	returns an Iterator object to traverse the collection's elements
`remove(`**`value`**`)`	finds and removes the given value from this collection

List<E> Methods (10.1)

`add(`**`index, value`**`)`	inserts given value at given index, shifting subsequent values right
`indexOf(`**`value`**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**`index`**`)`	returns the value at given index
`lastIndexOf(`**`value`**`)`	returns last index where given value is found in list (-1 if not found)
`remove(`**`index`**`)`	removes/returns value at given index, shifting subsequent values left
`set(`**`index, value`**`)`	replaces value at given index with given value
`subList(`**`from, to`**`)`	returns sub-portion at indexes **from** (inclusive) and **to** (exclusive)

Stack<E> Methods

`peek()`	returns the top value from the stack without removing it
`pop()`	removes the top value from the stack and returns it; `peek`/`pop` throw an `EmptyStackException` if the stack is empty
`push(`**`value`**`)`	places the given value on top of the stack

Queue<E> Methods

`add(`**`value`**`)`	places the given value at the back of the queue
`peek()`	returns the front value from the queue without removing it; returns `null` if the queue is empty
`remove()`	removes the value from the front of the queue and returns it; throws a `NoSuchElementException` if the queue is empty

Map<K, V> Methods (11.3)

`clear()`	remove all keys and values from the map
`containsKey(`**`key`**`)`	`true` if the map contains a mapping for the given key
`get(`**`key`**`)`	the value mapped to the given key (`null` if none)
`keySet()`	returns a `Set` of all keys in the map
`put(`**`key, value`**`)`	adds a mapping from the given key to the given value
`remove(`**`key`**`)`	removes any existing mapping for the given key
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`
`values()`	returns a `Collection` of all values in the map

String Methods (3.3, 4.4)

`charAt(`**`i`**`)`	the character in this String at a given index
`compareTo(`**`str`**`)`	compare strings by ABC order; returns <0 (less), 0 (=), or >0 (greater)
`contains(`**`str`**`)`	`true` if this String contains the other's characters inside it
`endsWith(`**`str`**`)`	`true` if this String ends with the other's characters
`equals(`**`str`**`)`	`true` if this String is the same as *str*
`equalsIgnoreCase(`**`str`**`)`	`true` if this String is the same as *str*, ignoring capitalization
`indexOf(`**`str`**`)`	first index in this String where given String begins (-1 if not found)
`lastIndexOf(`**`str`**`)`	last index in this String where given String begins (-1 if not found)
`length()`	number of characters in this String
`startsWith(`**`str`**`)`	`true` if this String begins with the other's characters
`substring(`**`i, j`**`)`	characters in this String from index *i* (inclusive) to *j* (exclusive)
`toLowerCase(), toUpperCase()`	a new String with all lowercase or uppercase letters

Random Methods (5.1)

`nextBoolean()`	random `true`/`false` result
`nextDouble()`	random real number between 0.0 and 1.0
`nextInt()`	random integer
`nextInt(`**`max`**`)`	random integer from 0 to *max* - 1, inclusive

```
public class ListNode {
    public int data;
    public ListNode next;
    public ListNode(int data) { ... }
    public ListNode(int data, ListNode next) { ... }
}
public class LinkedIntList {
    private ListNode front;
    methods
}

public class IntTreeNode {
    public int data;
    public IntTreeNode left;
    public IntTreeNode right;
    public IntTreeNode(int data) { ... }
    public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) {...}
}
public class IntTree {
    private IntTreeNode overallRoot;
    methods
}
```