Instructions

This exam contains 10 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn’t need to do this at all, so be careful when making assumptions.

You may avail yourself of the punt rule. If you write down punt for any part of a question with a specifically-assigned point value, you will earn 1/5 of the points for that question (rounded down).

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>
Question 1. Short Answer (25 points).

a. (5 points) Compare and contrast interfaces and abstract classes.

Answer:
Interfaces specify method names and their types, but do not include any code. A class may implement many interfaces. An abstract class, on the other hand, can contain some methods with code, as well as some abstract methods with no code. A class may only extend one other class, including at most one abstract class. In some sense, an interface is like an abstract class that contains only abstract methods.

b. (5 points) Explain why it’s almost always a bad idea to use the `String(String)` constructor to create a new String object.

Answer:
Because Strings are immutable, it’s safe to share objects that represent the same string. Using the `String(String)` constructor prevents sharing and thus wastes space to little or no benefit.

c. (5 points) Explain the difference between the declared and actual type of a variable. Which one is used in dynamic dispatch?

Answer:
The declared type of a variable is the type it’s given in the source code at compile time. The actual type of a variable is its run-time type, which is set once and for all at object creation time. An object’s actual type is used when determining which method to invoke in dynamic dispatch.
d. (10 points) Consider the following class:

```java
class MyList {
    ...
    // Inserts the specified element at position index in the list. index
    // must be >= 0 and <= size of the list. */
    public void insert(int index, Object element) { ... }
}
```

Suppose we want to subclass `MyList` and override `insert`. State whether each of the following extensions is safe, in the sense that someone expecting a `MyList` can be given a `MySubList` instead, and explain your answer briefly. (Assume that the code is valid and matches the specification.)

- public class MySubList extends MyList {
  /* Inserts the specified element at position index in the list. If index < 0
   or index > size of the list, throws IndexOutOfBoundsException (a RuntimeException). */
  public void insert(int index, Object element) throws IndexOutOfBoundsException { ... }
}

  Answer: This is not safe, because a client of `MyList` is not expecting an exception for this method.

- public class MySubList extends MyList {
  /* Inserts the specified element at position index in the list. Does
   nothing if index is not >= 0 and <= size of the list. */
  public void insert(int index, Object element) { ... }
}

  Answer: This is safe, because it has a less restrictive precondition.

- public class MySubList extends MyList {
  /* If the element is of the same class as the other elements in the list,
   inserts the specified element at position index in the list. Otherwise
   does nothing. In either case, index must be >= 0 and <= size of the list. */
  public void insert(int index, Object element) { ... }
}

  Answer: This is not safe, because it has a more restrictive precondition, requiring that the type of `element` match the type of the other elements in the list.
Question 2. Refactoring (25 points).

a. (15 points) In class we described a number of bad smells in code that suggest the need for refactoring. For each of the following bad smells, state which of the following refactorings you would use to “deodorize” them, and explain your answer: Pull Up, Extract Method, Introduce Parameter Object, Extract Class, and Replace Temp with Query.

You may use zero, one, or more than one refactoring per smell, and you may or may not use all of the above refactorings.

i. (5 points) B and C are both immediate subclasses of A, and buried in the code for methods B.foo() and C.bar() is a section of identical code.

Answer:

The problem here is duplicate code. First, pull out the identical sequences of code into identical methods using Extract Method. Then, apply Pull Up to move the identical method into class A.

ii. (5 points) Method foo() has 16 parameters.

Answer:

The problem here is having a large number of parameters. Use Introduce Parameter Object to group several of foo()’s parameters that are logically associated, and thereby to reduce the number of parameters to foo().

iii. (5 points) Method foo() is 2000 lines long.

Answer:

The problem here is having a (much, much too) long method. Use Extract Method to pull out logical chunks of code into other methods, and use Replace Temp with Query to replace some intermediate computations with method calls to reduce the size of the method.
b. (10 points) For each of the following two refactorings, describe any circumstances in which the refactoring is not safe, or argue that the refactoring is always safe.

i. (5 points) Encapsulate Downcast. Change a method that returns an Object that is downcast by the caller so that the downcast happens in the function. Example:

```java
Object lastReading() {
    return readings.lastElt();
}
⇒
Reading lastReading() {
    return (Reading) readings.lastElt();
}
```

Answer: This change is in general safe, since all clients expecting an Object can be given a subclass instead, with no change. However, the refactoring cannot be done if a parent or child class has a different version of lastReading that can't be changed in the same way.

ii. (5 points) Change Inner Class to Outer Class. Pull a static inner class out of an encapsulating outer class. Example (the right-hand code is in two files):

```java
class ObjStack {
    static class Node {
        Object elt;
        Node next;
        Node(Object o, Node n) {
            elt = o; next = n;
        }
    }
    ...
}
class Node {
    Object elt;
    Node next;
    Node(Object o, Node n) {
        elt = o; next = n;
    }
}
```

Answer: Because the inner class is static, it can be pulled out without worrying about holding a pointer to the enclosing class. However, if there are any private static fields of the outer class that are accessed by the inner class then we cannot perform the refactoring without making those classes public or protected. Also, we cannot perform this refactoring if its name conflicts with another class. Finally, if the inner class was private, then making it an outer class will render it publicly visible.
Question 3. Testing (20 points). Below is part of the code for the ShapeBuffer class from Project 2.

```java
public class ShapeBuffer {

    public static final int MAX_CAPACITY = 10; // max capacity of buffer
    private boolean dirty = false;
    private int minZ = Integer.MAX_VALUE;
    private int maxZ = Integer.MIN_VALUE;
    private Hashtable shapes; // our list of shapes
    private int lastID = 0;

    public ShapeBuffer() { shapes = new Hashtable(); }

    /**
    * This method adds a new encapsulated shape to the collection. It returns a
    * newly generated ID for the new shape. If a shape is successfully added, this method
    * sets the dirty flag.
    *
    * zIndex - The zIndex for the new shape.
    * data - The shape data. InvalidShapeException is thrown if data is null
    *
    * Returns - A numerical identifier for the newly added shape. The returned ID
    * obeys the following:
    *
    * 1. The ID for the first shape added is 1.
    * 2. Valid IDs are >= 1
    * 3. This method will return a valid ID unless an exception occurs
    * 4. For two consecutive calls to addShape, the returned IDs should be
    *     monotone (i.e. the earlier call returns a smaller ID than the second call).
    *
    * throws InvalidShapeException If the shape data is null.
    * throws OverflowException If the buffer’s maximum capacity was reached.
    */
    public int addShape(int zIndex, Object data) throws InvalidShapeException, OverflowException {
        int ID;
        EncapsulatedShapeImpl newShape;
        ID = lastID++;
        newShape = new EncapsulatedShapeImpl(ID, zIndex, data);
        shapes.put(new Integer(ID), newShape);
        if (minZ > zIndex) minZ = zIndex;
        if (maxZ < zIndex) maxZ = zIndex;
        dirty = true;
        return ID;
    }
}
```

6
a. (15 points) The `addShape` method on the previous page has exactly three errors in it. Write a JUnit class (just like in Project 2) that has three test methods that catch all the errors in `addShape`. Each test case should find exactly one error (they do not overlap). You may use any `assertX` method you choose (e.g., `assertTrue`). You may use other `ShapeBuffer` methods not listed in your tests, though you don’t need to, and you may use `setUp` and or `tearDown`.

```java
public class ShapeBufferTestCase extends TestCase {

    // FILL IN
}
```

b. (5 points) Do your tests above cover all statements in the `addShape` method? Justify your answer.

Question 4. Design Patterns (30 points). In this question, you will implement the rock-paper-scissors game using double-dispatch (part of the Visitor pattern). In this game, two players secretly choose rock, paper, or scissors and then reveal their choice simultaneously. If the two players choose the same item, they tie. Otherwise, rock beats scissors, paper beats rock, and scissors beats paper.

Below we’ve provided you with a main method that runs one round of the game. This method creates instances of the classes named on the command line and calls the accept method from the Choice interface. c0.accept(c1) should return -1 if c0 loses to c1, 0 if they tie, and 1 if c0 beats c1.

```java
public interface Choice {
    int versus(Rock r);
    int versus(Paper p);
    int versus(Scissors s);

    int accept(Choice c);  // Returns -1 if this loses to c, 0 if it ties c, 1 if it beats c
}

public class Main {
    public static Choice toChoice(String s) {
        if (s.equals("Rock")) return Rock.INSTANCE;
        else if (s.equals("Paper")) return Paper.INSTANCE;
        else if (s.equals("Scissors")) return Scissors.INSTANCE;
        else throw new IllegalArgumentException();
    }

    public static void main(String[] args) {
        Choice c0 = toChoice(args[0]);
        Choice c1 = toChoice(args[1]);

        int result = c0.accept(c1);

        System.out.print(c0.toString());
        switch (result) {
            case -1: System.out.print(" loses to "); break;
            case 0: System.out.print(" ties "); break;
            case 1: System.out.print(" beats "); break;
        }
        System.out.println(c1.toString());
    }
}
```
a (15 points). Implement the Rock class below. Your class must

- Have an accept method that behaves correctly according to the code for main. Your accept method should call versus.
- Use the Singleton pattern, with the single class instance accessible through a field named INSTANCE.
- Implement the Choice interface, shown on the previous page.

Your class must not use instanceof.

Note that you only need to write the Rock class, and we will grade your solution as if you had written the Paper and Scissors classes using the same pattern (i.e., an identical accept method and versus methods following analogous logic). However, it may help you to write out one of the other classes on scratch paper.

Hint: You do not need to write very much code for this problem. If you get stuck, don't forget to do part (b), which is independent of this part.

```java
public class Rock implements Choice {
    public String toString() { return "Rock"; }
    // FILL IN REST OF CLASS
```
b (15 points). Suppose that we want to implement a game with the same three items as part (a) but alternate rules. For example, we may want paper to beat scissors, scissors to beat rock, and rock to beat paper. We’d like to use the same Choice interface but have two sets of possible implementations: PlainRock, PlainPaper, and PlainScissors for the normal rules, and BizarroRock, BizarroPaper, and BizarroScissors for our new rules.

Below, draw a UML diagram showing this design using the Abstract Factory pattern. We’ve given you a head start. You should still use the Singleton pattern as well, so that there is exactly one instance each of PlainRock etc. You can use PR as an abbreviation for PlainRock, etc to save on writing.

You should show all classes in your diagram, but you need only provide as much detail in methods and fields as necessary for an intelligent programmer to figure out what’s happening. In particular, you don’t need to write pseudocode for every factory method—one will do. And you don’t need to list all the methods of a class if they’re just the ones shown in a superclass/interface. (So, for example, we don’t list the methods of Rock below.) You also don’t need to show pseudocode for the versus or accept methods, since you wrote those in part (a). Use underlining to indicate abstract classes and interfaces.

Finally, describe how the Main class will need to change to accommodate this design, assuming that which rules to use (Plain or Bizarro) is specified somehow on the command line.

```
Choice
  versus(Rock)
  versus(Paper)
  versus(Scissors)
  accept(Choice)

Rock
```