Name:

Midterm Sample Questions

Instructions

Here is a collection of sample midterm questions. These do not cover all topics on the midterm, and they might include material not on the midterm. The purpose is to give you some insight into the format of the test. This document is also not representative of the length of the midterm.

Also, problems 2 and 4 are way too long...

Question 1. Short Answer.

a. Compare and contrast interfaces and abstract classes.

> **Answer:** Interfaces specify method names and their types, but do not include any code or fields. 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. [Note: In recent versions of Java, interfaces actually can have code. So this is a bit of a stale question. But, for purposes of this class, interfaces won't have code in them.]

b. 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. 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 fixed at object creation time. An object's actual type is used when determining which method to invoke in dynamic dispatch.

d. Does having a `final abstract` method ever make sense? Explain.

> **Answer:** No. A `final` method cannot be overridden, and an `abstract` method has no implementation. Thus a `final abstract` method could never be implemented and is useless.

e. Describe two disadvantages of using reflection instead of regular Java operations. Explain your answers.

> **Answer:** There are four main disadvantages:
>
> 1. Readability – The same operations performed using reflection are much more verbose, and therefore much harder to read and understand.
> 2. Performance – Accessing fields or invoking methods with reflection is much slower than doing so directly. Since the use of reflection often gets in the way of compiler optimizations, it may be much, much slower.
> 3. Code size – Again, since performing the same task with reflection takes many more steps, it requires more code, which may increase the size of your program. (Note that good uses of reflection often reduce code size compared to large if/then/else block alternatives.)
> 4. Ease of errors – When invoking methods and accessing fields using reflection, many errors that would be caught be the compiler normally become run-time exceptions, making it easy to make certain mistakes.

Question 2. Double Dispatch. 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`.

```
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());
  }
}
```

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`.

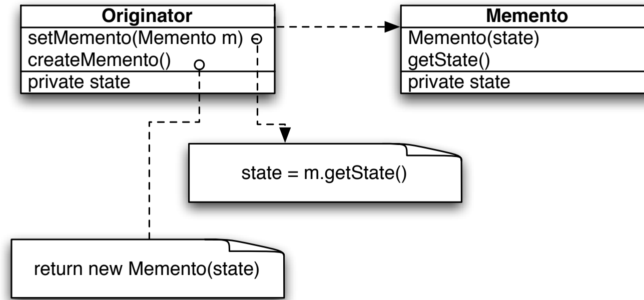
Note that you only need to write the `Rock` class, and we will grade your solution as if you had written the `Paper` and `Scissors` classes using the same pattern (i.e., an identical `accept` method and `versus` methods following analogous logic). However, it may help you to write out one of the other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

```
public class Rock implements Choice {
    public String toString() { return "Rock"; }
    // FILL IN REST OF CLASS
    private Rock() {}
    public static Rock INSTANCE = new Rock();

    public int versus(Rock r) { return 0; }
    public int versus(Paper p) { return 1; }
    public int versus(Scissors s) { return -1; }

    public int accept(Choice c) { return c.versus(this); }
}
```

Question 3. UML Diagrams. This question is about understanding UML diagrams. [Note: For this course, I don't actually care if you understand the details of UML diagrams, but I thought this would be useful practice with another pattern.] The *memento* design pattern is used to allow a third party to checkpoint internal state without having direct access to it. Below is the structure of this pattern (copied from Gamma et al). Here the Originator has some internal state that is private. A third party can save or restore that state with `createMemento` and `setMemento`.



Write Java code equivalent to the above structure diagram. That is, write Java code for the classes `Originator` and `Memento`. Your code must compile. The state field of the Originator should be a String. (*Hint:* If you understand the above diagram, this is an easy problem...don't think it's complicated.)

```
public class Originator {
    private String state;

    public void setMemento(Memento m) {
        state = m.getState();
    }

    public Memento createMemento() {
        return new Memento(state);
    }



}

public class Memento {

    private String state;

    public Memento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }



}
```

Question 4. Dynamic Proxies. A *security proxy* ensures that all clients that call methods of an object have permission to do so. Using dynamic proxies, on the next page implement the `createSecureObject()` method, which takes an Object `o` and a Permission `p`, and returns a new object `result`. The object `result` should implement all of the interfaces of `o`, **including** interfaces implemented by any superclass of `o`. (Note: This is slightly different than project 6.) Calls to `result.method(args)` should first call `checkPermission(p)` on the current security manager to see if access is allowed. If it's not, a `SecurityException` should be thrown. Otherwise `o.method(args)` should be invoked and its result returned. If `o.method(args)` throws an exception, that exception should be thrown (don't forget to unwrap it).

We've provided you with the necessary portion of the API below.

Class	Method	Description
Object	Class getClass()	Get the Class for this object
System	SecurityManager getSecurityManager()	Return the current Security Manager
SecurityManager	void checkPermission(Permission perm)	Throws a SecurityException if the requested access, specified by the given permission, is not permitted based on the security policy currently in effect. Otherwise has no effect.
Class	ClassLoader getClassLoader()	Return the class loader for this class
	Class[] getInterfaces()	Get the interfaces declared by this class
	Class getSuperclass()	Get the superclass of this class, or null if this class has no superclass
Method	Object invoke(Object obj, Object[] args) throws InvocationTargetException	Invokes the underlying method represented by this Method object, on the specified object with the specified parameters. (To simply things, we've eliminated some exceptions from the API.)
InvocationTarget-Exception	Throwable getCause()	Returns the cause of this exception (the thrown target exception, which may be null).
Proxy	static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h);	Returns an instance of a proxy class for the specified interfaces that dispatches method invocations to the specified invocation handler.
InvocationHandler	Object invoke(Object proxy, Method method, Object[] args) throws Throwable;	Processes a method invocation on a proxy instance and returns the result. This method will be invoked on an invocation handler when a method is invoked on a proxy instance that it is associated with.
Arrays	static List asList(Object[] a)	Returns a fixed-size list backed by the specified array.
LinkedList	void addAll(Collection c)	Appends all of the elements in the specified collection to the end of this list.
	Object[] toArray()	Returns an array containing all of the elements in this list in the correct order.

```
public class SecureObject implements Invocationhandler {

    private Object o;
    private Permission p;

    public SecureObject(Object o, Permission p) {
        this.o = o;
        this.p = p;
    }

    public static Object createSecureObject(Object o, Permission p) {
        Class c = o.getClass();
        List int = new LinkedList();
        for (Class i = c; i != null; i = i.getSuperclass())
            int.addAll(Arrays.asList(i.getInterfaces()));
        InvocationHandler h = new SecureObject(o, p);
        return Proxy.newProxyInstance(c.getClassLoader(),
            (Class[]) int.toArray(), h);

    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        try {
            System.getSecurityManager().checkPermission(p);
            return method.invoke(o, args);
        }
        catch (InvocationTargetException e) {
            throw e.getCause();
        }
    }
```