CS193J Handout #34

Winter, 2001-02

Final practice problems

The final exam will be at our regularly scheduled exam time: Thu Mar 21, 3:30-6:30 pm. The exam will be in Gates b01, which is down the hall from our regular room. We'll have a single alternate reading of the exam just before the regular time – email nick to sign up for the alternate.

SITN students: you have the two regular choices: 1) You are welcome to join us in person for the regular reading of the exam. This provides the most authentic and traditional college-exam experience. 2) You may wait at your site, and your coordinator can administer the exam there on Fri and fax it back (we are grading on Fri). SITN should have the exam for the coordinators by Fri morning. Your local SITN site coordinator should know how to deal with all this.

The final exam will cover all of the material we have seen this quarter. Most of the exam will be concerned with coding problems similar to the homeworks. Some of the problems will be essay/short-answer type on material from lecture. The short answer questions will be right=+1, blank=0, wrong=-1, so it will not be in your interest to guess if you don't know the answer.

The exam will be 3 hours long and will be open book, and open note, but you may not use a computer. As with any open-note exam, the questions will probably not be so simple that you can just look up the answer. Instead, the questions (like the homeworks) will require you to apply the concepts from lecture to solve small problems.

For a Stanford code-writing exam, familiarity with the topics is not enough. There is not enough time to re-learn the details during the exam. For a code writing exam, you need to be fluid and practiced with the code, relying on your notes only for the occasional detail. This requires practice. Fortunately, there are many sources of practice problems.

1. Understand all the example code from in lecture.

2. Understand your own solution code for each homework. The exam will include smaller but related code-writing problems. Understand the code well enough to write small solutions starting with a blank sheet of paper. On the exam, we will provide summaries of any necessary library message prototypes, so you do not need to memorize the interfaces to the library classes.

For code writing problems, we will not be especially picky about syntax or other basically conceptually shallow ideas. Instead, the questions will focus on understanding the core Java concepts.

How To Study For A Code Writing Exam

To study for code-writing exams, take a coding problem for which you have a solution available (lecture example, sample exam problem) and try to write the solution as you would on an exam **starting with a blank sheet of paper**. Trying to write the code, even with errors, will give you a much better understanding of the major concepts than a *passive* review of the solution code.

The rest of this handout includes a patchwork of questions from the last few years of exams (from Julie Zelenski and Jerry Cain) — I've selected questions which are most similar to this year's coverage. See our web page for a complete repository of old exams (but the older they are, the less relevant they are to our quarter.) Our exam will look a little different since we have covered somewhat different material, but this should give you a good idea of what a code writing exam looks like. On the real exam, each question will be on its own page so there will be room for you to write you solutions directly on the exam paper.

Material

The final will cover material from the entire quarter. The exam will focus on mainstream Java features and classes and will emphasize the material learned on the homework and presented in lecture. The sorts of things that you should be prepared to demonstrate knowledge of:

- Java syntax and language features— primitives, operators, control structures, arrays, Strings, Collections
- Objects and classes— instance variables and methods, access specifiers, "this", constructors and field initialization, method overloading, static and final modifiers
- Object-oriented design— associating behavior with the object
- Inheritance— subclassing, overriding, superclass/subclass structure, inheritance code factoring, polymorphism, abstract methods and classes, interfaces
- Inner classes— relationship to outer class, access specifiers, static versus non-static, anonymous inner classes
- Exceptions— try/catch/finally, throwing an exception, catching exceptions, checked vs unchecked, exception specifications on method declarations
- Input/output— text streams readers/writers, object streams for serialization
- Concurrency— Thread class, Runnable interface, scheduling issues, race conditions, mutual exclusion, synchronization, inter-thread communication via wait and notify
- Swing – layouts, JFrame, JComponent, paintComponent(), Graphics, control listeners, model-view-controller, swing threading, double buffering
- Networking—simple uses of URL and URLConnection
- XML – operating on the in-memory DOM
- Misc – Java performance, GC, java library areas

1) A new collection class

[Note: This question is not an exact match for this quarter, since we did not use the Hashtable class. However, the general structure of the question is similar to what would show up on an exam: take a class that was used in lecture or on a homework, and ask the students to write code for a variation on that class.]

The standard Java Hashtable manages a one-to-one mapping of keys to values. For this problem, you will implement a Multitable object that allows a one-to-many mapping where an entire vector of values is associated with each key. A Multitable could be used for a university database where the key was the class name and the associated values were the enrolled students or a datebook where the date was used as the key to retrieve people who were born on that day. For example:

```
Multitable bdays = new Multitable(); // map dates to people born that day

bdays.addValueForKey(new Date("10/29/80"), "Jill")); // Jill born 1/10/80
bdays.addValueForKey(new Date("11/14/76"), "Mati")); // Mati born 11/14/76
bdays.addValueForKey(new Date("11/14/76"), "Ling")); // add Ling, no replace

    // loop & print all names in table: Jill, Mati, and Ling (in some order)
Enumeration e = bdays.values();
while (e.hasMoreElements())
   System.out.println(e.nextElement());
```

Making optimal use of the standard built-in classes, the class's underlying storage is a Hashtable of entries using the key as given by the client and a Vector of elements as the associated value. Just as the ordinary Hashtable, the Multitable expects the key to respond properly to the hashCode() and equals() methods and can accept any type of Object as a value.

Although a full Multitable implementation would have many methods, you only need to write a subset. Your Multitable class must work for the client code given above. Thus, it needs a zero-argument constructor, an `addValueForKey` method that associates a new value with a key, and a `values` method that returns an Enumeration that iterates over all values for all keys in the table. The enumeration is free to visit the values in any order it chooses.

The add operation raises an `IllegalArgumentException` if the client tries to store a null key or null value. The Multitable enumeration raises a `NoSuchElementException` if asked for the next element when hasMoreElements() would return false.

A few notes:

- We are interested in proper use of access specifiers for this question, so do take care with them.
- The Enumeration object returned by values should be implemented as an inner/nested class.
- The client can choose to add duplicates (e.g. add “Mati” again on the same date), you do not need to take any special action to avoid storing duplicate values.
- Your Multitable should contain only a Hashtable of keys and values and no other duplicated storage of the keys or values.
- Both the addValueForKey() and values() method should run in constant time.
- Refer to the sample usage above to see the prototypes of the methods you need to match.

1 Solution) A new collection class

```
public class Multitable {
   private Hashtable table;
```

```
   public Multitable() {
      table = new Hashtable();
   }

   public void addElementForKey(Object key, Object value) {
      if (key == null || value == null)
         throw new IllegalArgumentException();
      Vector v = (Vector)table.get(key);
      if (v == null) {
         v = new Vector();
         table.put(key, v);
      }
      v.addElement(value);
   }

   public Enumeration elements() {
      return new MTEnumeration();
   }

   private class MTEnumeration implements Enumeration {

      Enumeration vectorEnum, tableEnum;

      MTEnumeration() {
         tableEnum = table.elements();
         vectorEnum = null;
      }

      public boolean hasMoreElements() {
         return (tableEnum.hasMoreElements() ||
            (vectorEnum != null && vectorEnum.hasMoreElements()));
      }

      public Object nextElement() {
         if (!hasMoreElements()) throw new NoSuchElementException();
         if (vectorEnum == null || !vectorEnum.hasMoreElements())
            vectorEnum = ((Vector)tableEnum.nextElement()).elements();
         return vectorEnum.nextElement();
      }
   }

}
```

2) Understanding Java code

```
public abstract class Vegetable {
    protected int roots = 7;

    public void eat() {
        slice();
        saute();
    }

    public static void saute() {
        System.out.println("Red");
    }

    public abstract void slice();
}

public class Tuber extends Vegetable {
    public Tuber() {
        mash();
    }

    public void slice() {
        System.out.println("Yellow");
        mash();
    }

    public void mash() {
        System.out.println("Blue " + roots);
    }
}
public class Potato extends Tuber {
    public Potato() {
        roots = 59;
    }

    public void eat() {
        if (this.equals(new Potato()))
            System.out.println("Orange");
        else
            System.out.println("Brown");
        super.eat();
    }

    public static void saute() {
        System.out.println("Aqua");
    }

    public void mash() {
        System.out.println("Pink " + roots);

    }
}
```

2 Solution) Understanding Java code

Consider the following method that is declared to take a `Vegetable` object as a parameter:

```
void binky(Vegetable veg) {
    veg.eat();
}
```

What are the possible types of objects that `veg` may be pointing to at runtime? For each possibility, trace through a call to the `binky` method and show the output that would be printed.

2 Solution

The parameter can be either of class Tuber or Potato. It cannot be a pure Vegetable since that is an abstract class and thus cannot be instantiated.

For a Tuber, the output is:

```
Yellow
Blue 7
Red
```

For a Potato, the output is:

```
Pink 7      // note the new Potato() that is created from within eat()
Brown       // default implementation of equals() only compares pointers
Yellow
Pink 59
Red                 // statics are always CT-bound
```

3) Inheritance

On the next page, you'll find code for the starting implementation of a generic Student class. The standard student responds to messages to study, eat, nap, and take an exam, and tracks the student's knowledge and energy level. Both of these are expressed as an integer (higher is better, 0 or negative is bad). For example, napping increases the student's energy by 2, studying decreases energy and increases knowledge based on the number of hours studied.

You are going to introduce three classes based on Student: Sleepy, Grumpy, and Happy. All three should understand all the same messages as Student and have the same general behavior of eating, napping, studying, taking exams and tracking their knowledge and energy.

We're interested in what happens when a student prepares for and takes one exam, simulated by sending a `void doOneExam(String subject)` message to a Student object. A student first must get prepared for the exam. An ordinary student feels prepared if their knowledge is 10 or greater or if their energy is 15 or greater. If the student already feels prepared, they go ahead and take the exam, otherwise they study some followed by a break to eat and nap, and then do another check to see if they feel prepared. The student iterates likes this, studying, eating, and napping, until they feel prepared, at which point, they take the exam. The student has one chance to pass the exam. Whether the student passes depends on their knowledge and energy and a bit of randomness, as shown in the code on the next page. If the student passes the exam, they blow off some steam by celebrating in their own way.

In addition to the standard behavior described above, the different students have some unique quirks of their own. Sleepy students are fond of napping. Whenever a Sleepy student naps, they nap twice as long as an ordinary student, thus increasing their energy by 4 instead of 2. As part of getting prepared for an exam, a Sleepy student does the usual exam preparation until ready and then takes one extra nap. What does a Sleepy student do to celebrate a successful exam? Nap, of course!

Grumpy students are grumpy because they work too hard and stress too much. Whenever you ask a Grumpy student to eat, they study for "CS193J" instead. Unlike an ordinary student, a Grumpy student only considers their knowledge when deciding if they are ready to take an exam (i.e. energy doesn't figure into it). A Grumpy student starts with 10 as their necessary level of knowledge, but each time the student takes an exam and doesn't pass, they raise the requirement by 1. So a Grumpy student who has previously failed 4 exams won't consider themselves prepared until their knowledge is at least 14. Grumpy students celebrate passing an exam by studying for "CS193K" to get ahead for next quarter.

Happy students love taking CS classes and they are exceptionally good at them. A Happy student prepares like an ordinary student, but they always manage to pass the exam for any CS class (i.e. any subject beginning with "CS") no matter what their knowledge or energy level. A Happy student celebrates passing an exam by eating.

Your job is to design and implement the three student classes and implement the `void doOneExam(String subject)` method for all students.

A few notes:

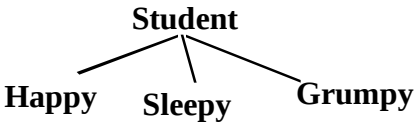
- You are free to add any other helper classes and can change or add to the generic Student class as well.
- We are not going to worry about allocation or initialization. You do not have to write any constructors. Where needed you can indicate the starting value for variables.
- You do not need to be concerned with access specifiers for this problem.
- **Your most important design goal is to avoid code duplication and place behavior in the correct classes**. It is recommended you think through the entire design before making any decisions.

Use this page to make modifications or additions to the Student class. You are free to change the code, add variables/methods, make the class abstract, etc. as desired. There are two blank pages after this for the other three student classes.

```
public class Student {

   public void eat() {
      energy++;
   }

   public void nap() {
      energy += 2;
   }

   public void study(String subject) {
      int numHours = (int)(Math.random()*5); // study up to 5 hours at a time
      knowledge += numHours;
      energy -= numHours;
      System.out.println("Spent " + numHours + " studying for " + subject);
   }

   public boolean didPass(String subject) {
      double probabilityOfPass = (knowledge*2 + energy)/30;
      return probabilityOfPass > Math.random(); // true if passed, false if not
   }

   public boolean takeExam(String subject) {
      System.out.println("Taking exam for " + subject);
      boolean passed = didPass(subject);
      System.out.println("Passed? " + passed);
      return passed;
   }

   protected int knowledge = 2;    // start with lots of studying to do
   protected int energy = 10;      // and some energy to burn
}
```

3 Solution) Inheritance

Obviously all the three students should be subclassed from Student. There are no special commonalties among the three classes that warrant any other intermediate classes:



Changes to base Student class:

Change the student class to be abstract.

Add these constants:

```
protected static final int knowledgeNeeded = 10, energyNeeded = 15;
```

Add these methods:

```
public void prepare(String subject) {
   while (!ready()) {
      study(subject);
      eat();
      nap();
```

```
      }
   }

   public boolean ready() {
      return (knowledge >= knowledgeNeeded || energy >= energyNeeded);
   }

   public void doOneExam(String subject) {
      prepare(subject);
      if (takeExam(subject))
         celebrate();
   }

   public abstract void celebrate();
}
```

Sleepy class:

```
public class Sleepy extends Student
{
   public void prepare(String subject) {
      super.prepare(subject);
      nap();
   }

   public void nap() {
      super.nap();   // nap twice what ordinary student does
      super.nap();
   }

   public void celebrate() {
      nap();
   }
}
```

Happy class:

```
public class Happy extends Student
{
   public boolean didPass(String subject) {
      return (subject.startsWith("CS") || super.didPass(subject));
   }

   public void celebrate() {
      eat();
   }
}
```

Grumpy class:

```
public class Grumpy extends Student
{
   protected int myKnowledgeNeeded = 10;

   public boolean takeExam(String subject) {
      boolean passed = super.takeExam(subject);
      if (!passed) myKnowledgeNeeded++; // if failed, study harder next time
      return passed;
   }
```

```
    public boolean ready() {
       return (knowledge >= myKnowledgeNeeded);
    }

    public void eat() {
       study("CS193J");
    }

    public void celebrate() {
       study("CS193K");
    }
}
```

4) Threads and callback interfaces

The MediaTracker is a convenient way to allow an activity (in this case, image loading) to proceed in a separate thread while providing the client with a means to check on its progress and block until the activity completes when necessary. This type of facility for handling tasks in parallel could be useful in many other contexts if designed with a more general "task tracking" approach.

To provide this, you will write the TaskTracker class. Once created, a TaskTracker will allow a client to pass off actions to the TaskTracker to be run in separate threads. The client will specify the action to be done in the new thread by passing an object that implements the Runnable interface. The client can run any kind of task (load an image, read a file, perform a database query, etc.) in a separate thread by implementing a Runnable to give to the TaskTracker.

Your job will be to write the TaskTracker class. There are two public methods you must include:

```
int doTaskInNewThread(Runnable clientTask)
void waitForID(int taskID)
```

(You may also need a public zero-arg constructor if the default one synthesized by the compiler is not sufficient).

The doTask method dispatches a new, separate thread to execute the client's task in parallel and immediately returns a unique integer that can be used to identify that task— for example, the first task might be given 0, each successful task the next higher number. ID numbers are never re-used.

The waitForID blocks until the completion of a previously dispatched task identified by its integer id. If the desired task has already finished, waitForID immediately returns, otherwise it will efficiently block the client's thread until the task thread completes and then return. The method should be designed to allow for the possibility that waitForID is called more than once per task and from more than one thread simultaneously.

A few notes:

- We are not concerned with access specifiers for this question, but we do care about proper use of synchronization.
- We are testing your knowledge of exceptions elsewhere, so you do not have to handle the case where the task given by the client is null or when waitForID is called with an inappropriate ID (i.e. an integer that was not previously returned from a call to doTask).

4 Solution) Threads and callback interfaces

a) There are a couple of different ways to solve this problem. The two most important requirements are that you properly synchronize so that you don't mistakenly assign the same id to two different tasks and to not busy-wait when blocking until a thread completes. Sleeping for a bit and then checking if the thread isAlive is busy-waiting. A much better strategy is to have the task signal back

via notify to the other threads who are waiting so they don't waste time checking in the meantime. You can wait and notify on the TaskTracker itself or on individual objects per thread to be even more efficient.

```
public class TaskTracker {

   // use a Vector of Boolean objects, one per id
   // value is false if not completed, true when thread finishes
   private Vector taskStatus = new Vector();

   public synchronized int startTask(final Runnable task) {
      final int taskId = taskStatus.size();  // use next index in vector
      taskStatus.addElement(new Boolean(false)); // entry starts as false
      new Thread(new Runnable() {      // wrap Runnable object
         public void run() {
            task.run();
            taskFinished(taskId); // after completing, update status
          }}).start();
      return taskId;
   }

   private synchronized void taskFinished(int id) {
      taskStatus.setElementAt(new Boolean(true), id); // set status to true
      notifyAll();   // alert all waiters that thread has finished
   }

   public synchronized void waitForTask(int id) {
      while (true) {
         if (((Boolean)taskStatus.elementAt(id)).booleanValue())
            return; // entry is vector is true, so task has finished
         try { wait();}    // wait til notified that a thread finished
         catch (InterruptedException e) {}
      }
   }
```

1) True/false (24 points)

[I've pared the questions down to the ones that make the most sense for our coverage this quarter. The questions may be true/false or short-answer. We will us a correct=+1, blank=0, wrong=-1 grading cheme for the short answer questions, so leave them blank if you don't know the answer.]

Answer true if the statement is always true, false otherwise. The value per question is small, so don't spend a lot of time agonizing over these. Each correct answer is +1, a wrong answer is -1 (to discourage guessing), and leaving it blank is 0 points. You can lose a lot of points if you are just guessing, so we recommend answering those that you know and leaving the others blank.

a) _____ You cannot instantiate a Java class which contains an abstract method.

true

c) _____ If a subclass constructor doesn't include a call to the superclass constructor, all of the inherited fields will be set to zero.

false, the default ctor of the superclass is called

e) _____ You don't need to synchronize multi-threaded access to a variable if it is only one line of code such as: `num++;`

`false`

f) _____ Static methods can throw exceptions.

true

k) _____ Private variables are not inherited by subclasses.

false

l) _____ Attempting to call another synchronized method from within a synchronized method on the same object will deadlock.

false, a thread can acquire the lock multiple times

m) _____ When serializing an object, all fields marked transient are skipped.

true

o) _____ The default implementation of the equals() method inherited from Object returns true when all the fields of the two objects exactly match.

False, the default just does ==

v) _____ In an instance method, it is legal to access the private fields of other objects of the same class as the receiver (i.e. not just the private fields of "this").

true

x) _____ An outer class can access its link to a non-static named inner class object using the syntax `InnerClassname.this`.

false, the outer object may have many inner objects, so this syntax doesn't even make sense

4) Threads and networking (24 points)

On the next page, you are given the starting implementation of a URLRace class that allows you to create a new race from an array of string URLs. When you ask the race object to run the race, it will dispatch separate threads, one to download each of the URLs. The threads download in parallel. When a thread finishes, it adds the URL to the end of the array of successful or failed URLs depending on the outcome. When all the threads have finished downloading, the successful and failed URLs are printed the order they completed from first to last.

The class as given is missing some of its implementation and has some errors in handling threads and synchronization. You are to finish its implementation and correct its problems.

The requirements are:

- You must fill in the implementation of the downloadURL() method which downloads the bytes from the specified URL. Don't worry about getting the content length in advance or fancy block reads, just call the simple single character read() method until you get -1 at EOF. The method returns the success of the operation. The method returns true if the URL was able

to be accessed and completely downloaded, false if any error occurred (malformed URL, can't connect to host, I/O problems, etc.)

- The class currently has no synchronization. It may be that some code passages need to be wrapped in synchronized blocks or entire methods may need to be marked synchronized. Find those areas that require synchronization and make the necessary changes for it to properly control access by multiple threads where needed. You should not overzealously lock large regions and inappropriately serialize the race.

- The runRace method currently busy-waits for all the download threads to finish before printing the results. This is wasteful and inefficient. Change the code to use wait and notify to allow for efficient waiting for the race to end before printing the results.

The code that is here compiles cleanly but may not behave correctly due to synchronization problems. You are free to change and augment any parts of this.

You can mark up the code below to indicate changes, but make it very clear what you are changing so that we understand your intentions. There is another blank page after this one for you to describe other changes you want to make.

```
public class URLRace
{
    protected String[] succeeded, failed;
    protected int numSucceeded, numFailed;

    protected void printResults()
    {
        System.out.println("Successes: ");
        for (int i = 0; i < nuSucceeded; i++)
           System.out.println((i+1) + ": " + succeeded[i]);
        System.out.println("Failures: ");
        for (int i = 0; i < numFailed; i++)
           System.out.println((i+1) + ": " + failed[i]);
    }

    public void runRace(String[] urlsToRace)
    {
        succeeded = new String[urlsToRace.length];
        failed = new String[urlsToRace.length];
        numSucceeded = numFailed = 0;
        for (int i = 0; i < urlsToRace.length; i++) {
           final String nextURL = urlsToRace[i];
           Thread t = new Thread(new Runnable() {
                public void run() { doOneURL(nextURL);}});
           t.start();        // start one thread per URL
       }
              // wait til all threads done
       while (numSucceeded + numFailed != urlsToRace.length)
              ;
       printResults(); // print the order they finished
    }


    protected void doOneURL(String url)
    {
        if (downloadURL(url))
           succdeded[numSucceeded++] = url;
        else
           failed[numFailed++] = url;
```

```
    }

    protected boolean downloadURL(String url) {} // needs implementation
}
```

4 Solution) Threads and networking (24 points)

Implement the downloadURL() method:

```
protected boolean downloadURL(String url)
{
   try {
      URL u = new URL(url);
      InputStream stream = u.openStream();
      while (stream.read() != -1) ;
      return true;
   } catch (Exception e) { // catch all errors: malformed, read, etc.
      return false;
   }
 }
```

Change the doOneURL() method to assign into the arrays using a synchronized block. We don't synchronize the entire method because we do not want to serialize the action of downloading! We also send out a notify when we finish.

```
protected void doOneURL(String url)
{
   boolean success = downloadURL(url);

   synchronized (this) {
      if (success)
         succeeded[numSucceeded++] = url;
      else
         failed[numFailed++] = url;
      notifyAll();  // within synchronized block on this
   }
}
```

Change the runRace() method to be synchronized and change the end of the method to do a proper wait for a notify signal:

```
while (numSucceeded + numFailed != urlsToRace.length) { // wait til done
   try { wait(); }
   catch (InterruptedException e) {}
}
printResults(); // print the order they finished
```

Alternatively, in a slightly better design, we could signal just once only from the last-finished thread and then just wait once in the runRace method.

Grading (mean for this question:17/24)
There was more variation in the responses to this question. Most were quite good, showing you learned a lot from LinkTester, but there were also some more troubling ones. Probably the most common error was forgetting that wait/notify can only be called on an object while within a synchronized context for that object. Other serious errors including not fixing the race condition when assigning to the arrays or serializing the entire race by synchronizing the entire doOneURL or downloadURL methods.