Question 1. (12 points) Proofs and correctness. One of the reasons for learning how to reason about programs is to be able to avoid or fix bugs. The following method is supposed to "rotate" the contents of an array left one position, with the leftmost array element moving to the right end. In other words, if the array initially contains the values {a,b,c,d,e}, the method should shift the elements to get {b,c,d,e,a}. Here is the code:

```
void rotateLeft(int[] a) {
   int temp = a[0];
   int k = 0;
   while (k < a.length) {
      a[k] = a[k+1];
   }
   a[a.length] = temp;
}
```

Give a proof below that this code works properly, or, if you discover any errors while trying to construct your proof, fix the code so that it works correctly and so that the proof shows that it does. Suggestions: use the notation `a[i..j]` to refer to sections of the array from `a[i]` to `a[j]` inclusive. You will also need to refer the original values of variables in your proof; use `x` and x_{pre} or similar notations to reference the current and original values of variables. You will need to define a suitable loop invariant and use it in your proof.

(There is additional blank space on the next page since your answer might not fit below.)

There are indeed bugs in the code. The most obvious is that `k` is not incremented inside the loop. Also, the code references `a[a.length]` as the last element of the array in two places, which would cause an `IndexOutOfBoundsException` when run. The correct index of the last element is `a[a.length-1]`.

We also need to add a precondition that the array has at least 1 element and is non-null, or else add additional code to handle those problems in order to show that the code is correct. In the solution we added a precondition rather than adding to the code and, since there is no formal specification of the method, that doesn't seem like an unreasonable solution.

Proof and fixed code on the next page.

Question 1. (cont.) Additional space for your answer if needed.

```
void rotateLeft(int[] a) {
   { pre: a!= null and a.length >= 1 }
   int temp = a[0];
   { temp == a[0]_pre }
   int k = 0;
   { inv: a[0..k-1] == a[1..k]_pre }
   while (k < a.length-1) {
      { a[0..k-1] == a[1..k]_pre && k < a.length-1 }
      a[k] = a[k+1];
      { a[0..k] == a[1..k+1]_pre && k < a.length-1 }
      k = k+1;
      { a[0..k-1] == a[1..k]_pre && k <= a.length-1 }
   }
   { inv && k == a.length-1 =>
     a[0..a.length-2] == a[1..a.length-1]_pre }
   a[a.length-1] = temp;
   { post: a[0..a.length-2] == a[1..a.length-1]_pre
           && a[length-1] == a[0]_pre }
}
```

Question 2. (12 points) Method madness. What output does this program produce when it is executed? (Note that the `main` method is at the bottom. This program does compile and run without errors.)

```
class A {
  void f()                 { System.out.println("Af"); }
  void f(String s)         { f(1,s); }
  void f(String s, int n)  { System.out.println("Afsn: "+s+n); }
  void f(int n, String s)  { System.out.println("Afns: "+n+s); }
  void f(int n)            { System.out.println("Afn: "+n); }
}

class B extends A {
  void f(int n)             { System.out.println("Bfn: " + n); }
  void f(String s, int n)   { System.out.println("Bfsn: "+s+n); }

}

class C extends B {
  void f(int n, String s)  { System.out.println("Cfns: "+n+s); }
  void f(int n)            { f("hello", n); }
}

public class Methods {
  public static void main(String[] args) {
    B b = new B();
    b.f();
    b.f(17);
    A c = new C();
    c.f(" hi ");
    c.f(331);
    c.f(17," question ");
    c.f(" answer ", 42);
  }
}
```

Output:

Af

Bfn: 17

Cfns: 1 hi

Bfsn: hello331

Cfns: 17 question

Bfsn: answer 42

Question 3. (8 points) Testing. Java, like most programming languages and the underlying computer hardware, implements integer division by truncating non-integer quotients towards zero. For example:

7/3 = 2 (-7)/3 = -2 7/(-3) = -2 (-7)/(-3) = 2.

The remainder operator % is defined as usual so that (*a*/*b*)**b* + (*a*%*b*) = *a*, and we always have that –*b* < *a*%*b* < *b*. Examples:

7%3 = 1 (-7)%3 = -1 7%(-3) = 1 (-7)%(-3) = -1

A consequence of the definitions is that the sign of the remainder is always the same as the sign of the numerator (*a* in *a*%*b*).

In mathematics, integer division and remainder often have a different definition, where division rounds down towards more negative numbers and the remainder is always positive. If we denote this division by div and remainder by rem, we have:

7 div 3 = 2	(-7) div 3 = -3	7 div (-3) = -2	(-7) div (-3) = 3
7 rem 3 = 1	(-7) rem 3 = 2	7 rem (-3) = 1	(-7) rem (-3) = 2

For the rem (sometimes called mod) operation, we always have 0 <= *a* rem *b* < *b*, and we still have the same identity as with truncating division: (*a* div *b*) * *b* + (*a* rem *b*) = *a*.

A colleague has implemented a Java method to compute the rem function as follows, using the Java % operator as part of the implementation:

```
/** return a rem b */
int rem(int a, int b) {
   int ans = a % b;
   if (ans < 0) ans = ans + abs(b);
   return ans;
}
```

Describe four black-box tests for this method. Each test should be intended to exercise a different possible subdomain, and you must briefly describe what that intended subdomain is for each test. For example, here is one test and description (no, you can't repeat this one):

Verify that rem(7,3) == 1. This is a basic test of the subdomain of positive integers.

(You may remove this page if convenient.)

(Note: there was a bug in the `rem` code in the original version of this question. But that actually has no effect on the answers, since the question asks about how to test the method. A good test suite would have a test that is revealing for any bug(!).)

Question 3. (8 points) List your four black-box tests for the `rem` method below.

(The question should have been worded to require original answers and disallow using the examples in the question as tests, but we gave credit for answers that did that since we didn't rule it out.)

Here are a few possible tests and domains using just positive arguments. Of course a thorough set of tests would also need to test many other inputs, particularly negative numbers.

Verify that rem(0,3) == 0. Test remainder 0 if quotient is 0.

Verify that rem(2,3) == 2. Test with quotient 0 and remainder non-zero.

Verify that rem(3,3) == 0. Test with quotient 1 and zero remainder.

Verify that rem(4,3) == 1. Test with quotient 1 and non-zero remainder.

Verify that rem(9,3) == 0. Test with quotient > 1 and zero remainder.

Verify that rem(10,3) == 1. Test with quotient > 1 and non-zero remainder.

Question 4. (6 points) Most programming environments have powerful debugging tools (eclipse, netbeans, jdb, gdb), and these can be quite helpful in isolating and understanding bugs. Yet many people still use `println` statements (or the equivalent) as a debugging technique. Give two good reasons for debugging using print statements instead of using a debugger. ("I don't want to learn how to use the debugger" or "I don't know how" or "I don't have time" are *not* good reasons.) Your answers should describe situations where printing is the debugging technique of choice, even after you become proficient with a debugger.

Here are several possible reasons for using print statements.

- **A print log allows you to see multiple moments in time at once and compare them in a text editor or otherwise.**
- **Print statements can be used to capture information for later examination when the program is in the field.**
- **Print statements may be the only way to capture information about timing-dependent errors, which can be hard or impossible to reproduce otherwise.**
- **Print statements may have less impact on timing-dependent code in the program compared to pausing in a debugger, which may affect program behavior.**

Question 5. (8 points) The typing rules for arrays in Java say that array types are *covariant* with respect to their element types. In other words, if `S` is a subtype of `T`, then array of `S` (`S[]`) is a subtype of array of `T` (`T[]`). For example, since `Integer` is a subtype of `Number`, the type `Integer[]` is a subtype of `Number[]`.

(a) (3 points) Give an example showing how this rule creates an unsound type system. That is, give an example that is legal using this subtyping rule for arrays, but causes incorrect behavior if execution is allowed to proceed with no additional runtime tests.

The following code is legal using the covariant array subtyping rule, even though it incorrectly stores a non-`Integer` value in an `Integer` array:

```
Integer[] ia = new Integer[10];
Number[] na = ia;
na[0] = 3.14;
```

(b) (3 points) The implementation can prevent these errors by inserting appropriate code to check for problems and throw exceptions if they occur during execution. Give a precise description of the check(s) that need to be done to avoid the problem(s).

On any assignment to an array element `a[i]=e`, a runtime check needs to be performed to verify that `e` can be assigned to a variable of the actual element type of the array object referenced by `a`.

(Java implementations insert this check and throw an `ArrayStoreException` if a violation occurs.)

(c) (2 points) Why did the Java designers include this covariant array typing rule in the language? They knew it was unsound and required additional runtime checks to avoid errors, but did it anyway. Why? (Be brief)

Java did not include generic types originally and this array subtyping rule allowed arrays of type `Object[]` to hold arbitrary collections of items, something that was needed to implement collection classes like `ArrayList`.

Question 6. (10 points) A generic warmup. The following class stores a list of integer values and contains a method to return the smallest value in a non-empty list.

Make changes and cross out old code as needed to turn this into a generic class that can be used to store lists of any element type `E` as long as those elements can be compared for ordering (i.e., any type `E` whose elements support the `compareTo` method). You can assume the code is adequate otherwise – you don't need to, for example, add additional tests to guard against `null` values.

Additions and changes shown in bold green type below. Deleted items are ~~crossed out.~~

```
public class MinList <E extends Comparable<E>> {
  List<~~Integer~~ E> items;
  /** Construct new empty list */
  public MinList() {
    items = new ArrayList<~~Integer~~ E>();
  }
  /** add new item x to list  */
  public void add(~~int~~ E x) {
    items.add(x);
  }
  /** return smallest value in a non-empty list */
  public ~~int~~ E minItem() {
    assert items.size() > 0;
    ~~int~~ E minValue = items.get(0);
    for (~~int~~ E val: items) {
      if (~~val < minValue~~ val.compareTo(minvalue) < 0) {
        minValue = val;
      }
    }
    return minValue;
  }
}
```

Question 7. (10 points) More generic things. Suppose we have the following classes:

```
class Animal extends Object { ... }
class Pet extends Animal { ... }
class Cat extends Pet { ... }
class Dog extends Pet { ... }
```

Now suppose we have a program that contains the following variables:

```
Animal a;                    List <? extends Pet> lep;
Pet p;                       List <? super Pet> lsp;
Cat c;
Dog d;
```

For each of the following, circle OK if the assignment has the correct types and will compile without errors. Circle Error if there is a type error. (Hint: think about which kinds of `List` types can be assigned to `lep` and `lsp`, and what types of values can be stored in those lists.)

(a) OK (Error) `lep.add(a);`

(b) OK (Error) `lep.add(p);`

(c) OK (Error) `lep.add(c);`

(d) (OK) Error `a = lep.get(0);`

(e) (OK) Error `p = lep.get(0);`

(f) OK (Error) `c = lep.get(0);`

(g) OK (Error) `lsp.add(a);`

(h) (OK) Error `lsp.add(c);`

(j) OK (Error) `a = lsp.get(0);`

(k) OK (Error) `c = lsp.get(0);`

Question 8. (14 points) MVC and Java GUIs. This question concerns the execution of the tiny model-view-controller application whose code appears on the next two pages. Briefly, when this program is executed, it displays a window on the screen with a circle in it. The main program (the controller) then reads commands from `System.in`. The commands "hide" and "show" cause the circle to disappear or reappear, and "quit" causes the program to terminate.

For this question, write down the sequence of method calls in the order they occur when the user enters the command "**show**". You should list all of the methods in the code that are executed *and* all of the methods called directly by those methods, including library functions. Your list should also include any callbacks triggered by any of the actions performed by any of these methods (i.e., if some method `foo` is called as a result of the methods executed to process "show", you should include it in your list in the appropriate place). Your list should start with `showShape`, called from `main` when a "show" command is read, and should stop after all of the methods executed directly or indirectly by that call have been listed. You should assume that nothing else is going on while the "show" command is being handled – i.e., there are no other system actions that would cause method calls or callbacks other than the ones caused by the code in the application.

In addition to listing the methods that are executed in the order that they are called, you should circle "pgm" if the method call is executed in the main program thread, or circle "UI" if the method call is executed by the Swing/AWT User Interface thread. The first method that is called, `showShape`, is given for you.

You may not need all of the blank spaces provided.

1. `showShape` (pgm) UI
2. `notifyViewer` (pgm) UI
3. `repaint` (pgm) UI
4. `paintComponent` pgm (UI)
5. `super.paintComponent` pgm (UI)
6. `clearRect` pgm (UI)
7. `isVisible` pgm (UI)
8. `setColor` pgm (UI)
9. `fillOval` pgm (UI)
10. _____ pgm UI
11. _____ pgm UI
12. _____ pgm UI

(You may remove the code listings on the next two pages if that is convenient.)

Question 8. (cont.)

```
/** Sample MVC application for CSE 331 final, 13wi */

import java.util.*;
import java.awt.*;
import javax.swing.*;


/** Create and control a tiny Swing MVC application */
public class ExamMVC {
  public static void main(String[] args) {
    // create model and view and hook them together
    ExamModel model = new ExamModel();
    ExamView view = new ExamView(model);
    model.setViewer(view);

    // set up window on screen, pack, and make visible
    JFrame frame = new JFrame("Annoying Final Exam Question");
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    frame.add(view);
    frame.pack();
    frame.setVisible(true);

    // read commands from user and process until quit entered
    String command;
    Scanner in = new Scanner(System.in);
    do {
      command = in.next();
      if (command.equals("show"))
        model.showShape(true);
      else if (command.equals("hide"))
        model.showShape(false);
    } while (!command.equals("quit"));

    // clean up
    frame.dispose();
  }
}


/** Model.  Store a mutable boolean value that represents whether a shape should be
  * displayed and notify viewer when the state might have been changed. */
class ExamModel {
  // instance variable - true if shape should be visible
  private boolean visible;

  // The viewer to notify when model events happen
  private ExamViewer viewer;

  /** construct new ExamModel with no viewer and initially visible */
  public ExamModel() {
    viewer = null;
    visible = true;
  }

  /** set viewer for this model to v */
  public void setViewer(ExamViewer v) { viewer = v; }

  /** return value of model state */
  public boolean isVisible() { return visible; }

  /** set visibility state */
  public void showShape(boolean isVisible) {
    visible = isVisible;
    if (viewer != null) {
      viewer.notifyViewer();
    }
  }
}
```

Question 8. (cont.)

```
/** Interface for viewers */
interface ExamViewer {
  /** called to notify viewer something interesting might have happened */
  public void notifyViewer();
}


/** Viewer - display a blank panel.  Include a circle in the middle if
  * the model state indicates the circle should be visible */
class ExamView extends JPanel implements ExamViewer {
  /** model we are watching */
  private ExamModel model;

  /** construct new ExamView JPanel watching model m with preferred size 100x100 */
  public ExamView(ExamModel m) {
    model = m;
    setPreferredSize(new Dimension(100,100));
  }

  /** react to any changes in model */
  public void notifyViewer() {
    repaint();
  }

  /** Repaint this panel and include centered circle if it should be visible */
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.clearRect(0,0,100,100);
    if (model.isVisible()) {
      g.setColor(Color.green);
      g.fillOval(10,10,80,80);
    }
  }
}
```

Question 9. (4 points) Usability. In lecture we listed these dimensions of usability: learnability, efficiency (of using the system), memorability, errors (rare and recoverable), satisfaction (is it enjoyable to use). Unfortunately in real systems it's sometimes difficult to achieve all of these goals simultaneously.

But how can that be? How is it that these worthy goals can be in conflict? Give a brief explanation using concrete details from a system you know (OS, application program, development environment, etc.). Keep your answer brief, and it will help (but is not required) to pick an example that the graders are likely to have some knowledge of.

A classic example is the Unix/Linux command-line interface. It is very efficient and enjoyable to use for proficient, expert users, but it is difficult to learn, cryptic, and hard to remember for beginners and occasional users.

Many systems have similar tensions between ease of use for trained experts, but difficult to learn or use if one is a novice. Also, if not done carefully, a system that is designed to be easy to learn can be clumsy to use, requiring experts to navigate simple-to-understand but tedious interfaces to get anything done.

Question 10. (6 points) UI Prototyping. The design community suggests using simple, low-fidelity prototypes like paper or postit notes for initial testing and evaluation of designs. Give two advantages of paper prototypes over building an executable prototype that runs on a computer or other hardware (be brief – a short phrase might be enough)

Some reasons:

- **Much cheaper and faster to produce compared to software development.**
- **Minimal cost to discard and modify when problems are discovered.**
- **Faster to produce and test multiple designs. Makes it feasible to explore a wider range of alternatives.**
- **Avoids temptation to continue with a bad design because so much time, effort, and resources have been invested in the current prototype.**

Question 11. (10 points) Design patterns. For each of the following design patterns, give an example that shows *one* benefit of using that particular pattern. It is not enough to describe what the pattern does; you need to explain why this is worthwhile. Example:

(not full credit) Singleton – makes it so there is only a single instance of a class.
(credit – two possibilities) Singleton – ensure correctness of a program using random numbers by ensuring there is only one random number generator object in the program. – or – ensure there is only one instance of a large object to conserve memory.

(a) Adapter

Allows existing code to be reused in different contexts with a different interface (API). Eliminates the need to change or reimplement code for the new situation.

(b) Builder

Simplifies complexity of code needed to construct new objects of a class with lots of configuration choices or optional parameters.

(c) Factory

Hides details and decisions about object creation from other parts of the program. Makes it possible to switch implementations or provide specialized subclasses by changing code in the Factory without affecting the rest of the program.

(d) Proxy

Controls access to other objects. The local (proxy) object can stand in for some other object, providing features like remote access over a network, security, locking, etc.

(e) Model-View-Controller

Separates data and algorithms from user interfaces. Reduces coupling between these parts and makes it easy to add or change user interfaces without needing to alter the core of the application.

Have a great spring break! See you soon!!