Homework 7

Due 24 November

Handout 21
CS242: Autumn 2004
17 November

Reading

1. Chapter 13, Java.
2. The web page at hhttp://java.sun.com/j2se/1.5.0/docs/guide/language/index.html summarizes the main new features of Java 1.5 and provides links to additional information. The one-page (or slightly longer) descriptions reached through the links at the top of the page give about the right amount of information for this course.
3. *Optional:* The web page at http://java.sun.com/developer/technicalArticles/releases/j2se15/ provides breif summaries of Java 1.5 features (at a level appropriate to this course) and also provides links to the JSRs at the end of the page. The JSRs provide more details than we will cover in this course.

Problems

1. .... Java Interfaces and Multiple Inheritance

In C++, a derived class may have multiple base classes. In contrast, a Java derived class may only have one base class but may implement more than one interface. This question asks you to compare these two language designs.

(a) Draw a C++ class hierarchy with multiple inheritance using the following classes:

Pizza, for a class containing all kinds of pizza,
Sausage, for pizza that has sausage topping,
Ham, for pizza that has ham topping,
Pineapple, for pizza that has pineapple topping,
Mushroom, for pizza that has mushroom topping,
Hawaiian, for pizza that has ham and pineapple topping,
MeatLover, for pizza that has ham and Sausage topping,
Supreme, for pizza that has everything

(b) If you were to implement these classes in C++, for some kind of pizza manufacturing robot, what kind of potential conflicts associated with multiple inheritance might you have to resolve?

(c) If you were to represent this hierarchy in Java, which would you define as interfaces and which as classes? Write your answer by carefully redrawing your picture, identifying which are classes and which are interfaces. If your program creates objects of each type, you may need to add some additional classes. Include these in your drawing.

(d) Give an advantage of C++ multiple inheritance over Java classes and interfaces and one advantage of the Java design over C++.

2. .... Array Covariance in Java

Java array types are covariant with respect to the types of array elements (i.e., if `B` <: `A`, then `B[]` <: `A[]`). This can be useful for creating functions that operate on many types of arrays. For example, the following function takes in an array and swaps the first two elements in the array.

```
1:   public swapper (Object[] swappee){
2:      if (swappee.length > 1){
3:        Object temp = swappee[0];
4:        swappee[0] = swappee[1];
5:        swappee[1] = temp;
6:      }
7:   }
```

This function can be used to swap the first two elements of an array of objects of any type. The function works as is and does not produce any type errors at compile-time or run-time.

(a) Suppose `a` is declared by `Shape[] a` to be an array of shapes, where `Shape` is some class. Explain why covariance (if `B` <: `A`, then `B[]` <: `A[]`) allows the type checker to accept the call `swapper(a)` at compile time.

(b) Suppose `Shape[] a` as in part (a). Explain why the call `swapper(a)` and execution of the body of `swapper` will not cause a type error or exception at run time.

(c) Java uses run-time checks, as needed, to make sure that certain operations respect the Java type discipline at run time. What run-time type checks occur in the compiled code for the `swapper` function and where? List the line number(s) and the check that occurs on that line.

(d) A friend of yours is aghast at the design of Java array subtyping. In his brilliance, he suggests that Java arrays should follow contravariance instead of covariance (i.e., if `B` <: `A`, then `A[]` <: `B[]`). He states that this would eliminate the need for run-time type checks. Write three lines of code or less that will compile fine under your friend's new type system, but will cause a run-time type error (assuming no run-time type tests accompany his rule). You may assume you have two classes, A and B, that B is a subtype of A, and that B contains a method, `foo`, not found in A. We give you two declarations that you can assume before your three lines of code.

```
B b[];
A a[] = new A[10];
```

(e) Your friend, now discouraged about his first idea, decides that covariance in Java is alright after all. However, he thinks that he can get rid of the need for run-time type tests through sophisticated compile time analysis. Explain in a sentence or two why he will not be able to succeed. You may write a few lines of code similar to those in part (d) if it helps you make your point clearly.

3. .... Exceptions, Memory Management, and Concurrency

This question asks you to compare properties of exceptions in C++ and Java.

(a) In C++, objects may be reside in the activation records that are deallocated when an exception is thrown. As these activation records are deallocated, all of the destructors of these stack objects are called. Explain why this is a useful language mechanism.

(b) In Java, objects are allocated on the heap instead of the stack. However an activation record that is deallocated when an exception is raised may contain a pointer to an object. If you were designing Java, would you try to call the finalize methods of objects that are accessible in this way? Why or why not?

(c) Briefly explain one programming situation in which the C++ treatment of objects and exceptions is more convenient than Java and one situation in which Java is more convenient than C++.

(d) In languages that allow programs to contain multiple threads, several threads may be created between the point where an exception handler is established and the point where an exception is thrown. In the spirit of trying to abort any computation that is started between these two points, a programming language might try to abort all such threads when an exception is raised. In other words, there are two possible language designs:

- raising an exception only affects the current thread, or
- raising an exception aborts all threads that were started between the point where the exception handler is established and the point where the exception is thrown.

Which design would be easier to implement? Explain briefly.

(e) Briefly explain one programming situation in which you would like raising an exception to abort all threads that were started between the point where the exception handler is established and the point where the exception is thrown. Can you think of a programming situation where you would prefer not to have these threads terminated?

4. .... Stack Inspection

One component of the Java security mechanism is called *stack inspection.* This problem asks you some general questions about activation records and the run-time stack then asks about an implementation of stack inspection that is similar to the one used in Netscape 3.0. Some of this problem is based on the book *Securing Java*, by Gary McGraw and Ed Felten.

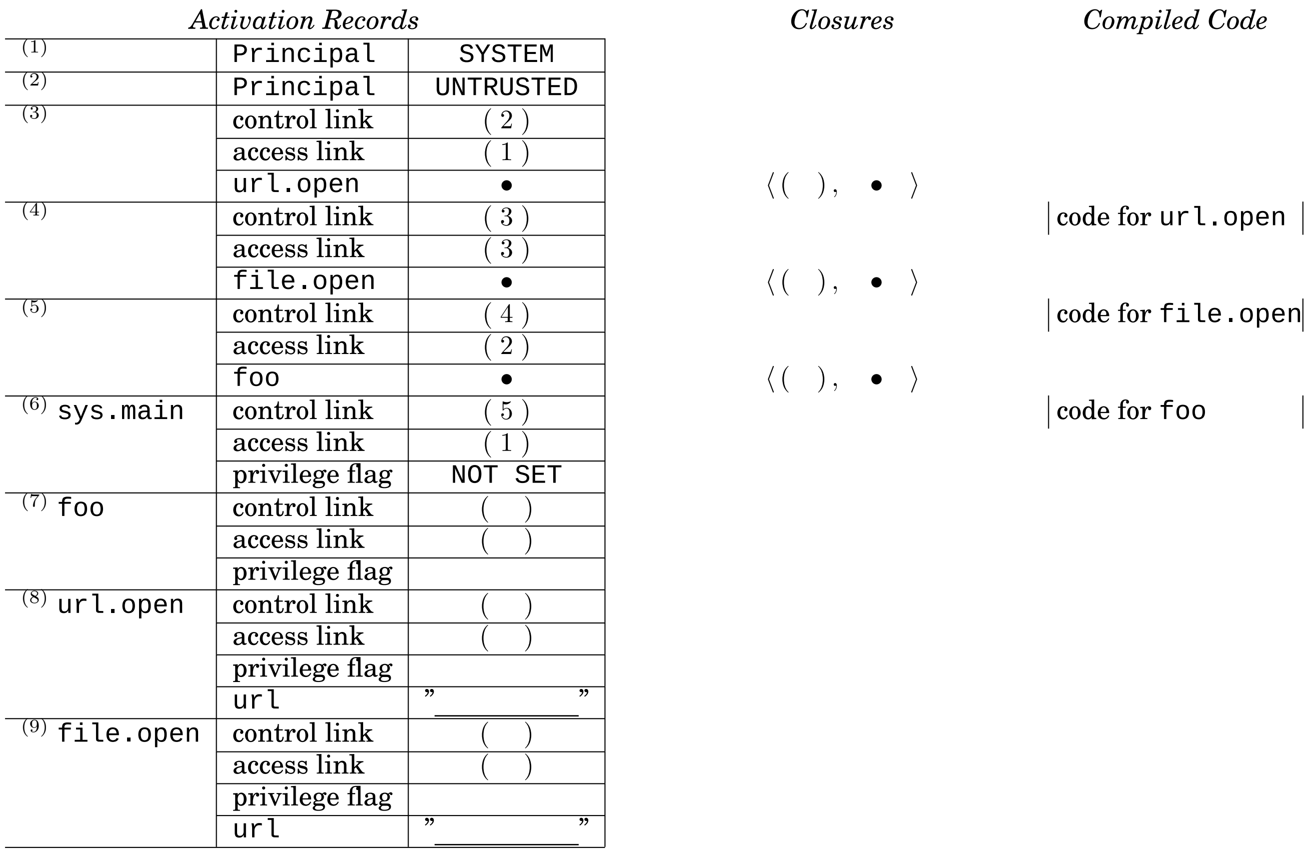
Parts of this problem will ask about the following functions, written in a Java-like pseudocode. In the stack used in this problem, activation records will contain the usual data (local variables, arguments, control and access links, etc.) plus a *privilege flag*. The privilege flag is part of our security implementation and will be discussed later. For now, we will just mention that `SetPrivilegeFlag()` sets the privilege flag for the current activation record.

```
void url.open(string url) {
    int urlType = GetUrlType(url); // Gets the type of URL

    SetPrivilegeFlag();

    if (urlType == LOCAL_FILE)
        file.open(url);
}
void file.open(string filename) {
    if (CheckPrivileges())
    {
        // Open the file
    }
    else
    {
        throw SecurityException;
    }
}
void foo() {
    try {
        url.open("confidential.data");
    } catch (SecurityException) {
        System.out.println("Curses, foiled again!\n");
    }
    // Send file contents to evil competitor corporation
}
```

(a) Assume that the URL `confidential.data` is indeed of type `LOCAL_FILE`, and that `sys.main` calls `foo()`. Fill in the missing data in the following illustration of the activation records on the run-time stack just before the call to `CheckPrivileges()`. For convenience, ignore the activation records created by calls to `GetUrlType()` and `SetPrivilegeFlag()`. (They would have been destroyed by this point anyway.)



Activation Records

(1)	`Principal`	`SYSTEM`
(2)	`Principal`	`UNTRUSTED`
(3)	control link	(2)
	access link	(1)
	`url.open`	•
(4)	control link	(3)
	access link	(3)
	`file.open`	•
(5)	control link	(4)
	access link	(2)
	`foo`	•
(6) `sys.main`	control link	(5)
	access link	(1)
	privilege flag	`NOT SET`
(7) `foo`	control link	( )
	access link	( )
	privilege flag	
(8) `url.open`	control link	( )
	access link	( )
	privilege flag	
	`url`	” ________ ”
(9) `file.open`	control link	( )
	access link	( )
	privilege flag	
	`url`	” ________ ”

(b) As part of stack inspection, each activation record is classified as either `SYSTEM` or `UNTRUSTED`. Functions that come from system packages are marked `SYSTEM`. All other functions (including user code and functions coming across the network) are marked `UNTRUSTED`. `UNTRUSTED` activation records are not allowed to set the privilege flag.

Effectively, every package has a global variable `Principal` which indicates whether the package is `SYSTEM` or `UNTRUSTED`. Packages which come across the network have this variable set to `UNTRUSTED` automatically on transfer. Activation records are classified as `SYSTEM` or `UNTRUSTED` based on the value of `Principal`, which is determined according to static scoping rules.

List all activation records (by number) that are marked `SYSTEM` and list all activation records (by number) that are marked `UNTRUSTED`.

(c) `CheckPrivileges()` uses a dynamic-scoping approach to decide whether the function corresponding to the current activation record is allowed to perform privileged operations. The algorithm looks at all activation records on the stack, from most recent on up, until:

- It finds an activation record with the privilege flag set. In this case it returns `TRUE`. Or,
- It finds an activation record marked `UNTRUSTED`. In this case it returns `FALSE`. (Remember that it is not possible to set the privilege flag of an untrusted activation record.) Or,
- It runs out of activation records to look at. In this case it returns `FALSE`.

What will `CheckPrivileges()` return for the stack shown above (resulting from the call to `foo()` from `sys.main`? Please answer “True” or ”False.”

(d) Is there a security problem in this code? (I.e., will something undesirable or “evil” occur when this code is run?)

(e) Suppose that `CheckPrivileges()` returned `FALSE` and thus a `SecurityException` was thrown. Which activation records from part (a) will be popped off the stack before the handler is found? List the numbers of the records.

5. .... Java Bytecode Analysis

Java programs are compiled into bytecode, a simple machine language that runs on the Java Virtual Machine. Note that it is possible that bytecode could be written by hand, or that compiled bytecode get corrupted when transmitted over the network. So, when a class is loaded by the JVM, it is first examined by the bytecode verifier. The verifier performs static analysis of the class to ensure that a program will not cause an unchecked runtime type error.

One kind of bytecode error that the verifier should catch is use of an uninitialized variable. Here is a code fragment in Java and some corresponding bytecode. We have added comments to the bytecode to help you figure out the effects of the instructions.

```
// Java Source Code
Point p = new Point(3);
p.Print();
```

```
// Compiled Bytecode
1: new #1 <Class Point>                    // allocate space for Point
2: dup                                     // duplicate top entry on stack
3: iconst 3                                // push integer 3 onto stack
4: invokespecial #4 <Method Point(int)>    // invoke constructor, which pops
                                           //   its argument (3) and one of
                                           //   the pointers to p off stack
5: invokevirtual #5 <Method void Print()>  // invoke print method, popping
                                           //   the other pointer to p.
```

The first line of the Java source allocates space for a new `Point` object and calls the `Point` constructor to initialize this object. The second line invokes a method on this object and therefore can be allowed only if the object has been initialized. It is easy to verify from this Java source that `p` is initialized before it is used.

Checking that objects are initialized before use in the bytecode is more difficult. The Java implementation creates objects in several steps: First, space is allocated for the object. Second, arguments to the constructor are evaluated and pushed onto the stack. Finally, the constructor is invoked. In the bytecode for this example, the memory for `p` is allocated in line 1, but the constructor isn't invoked until line 4. If several objects are passed as arguments to the constructor, there could be an even longer code fragment between allocation and initialization of an object, possibly allocating multiple new objects, duplicating pointers, and taking conditional branches.

To account for pointer duplication, some form of aliasing analysis is needed in the bytecode verifier. This problem will consider a simplified form of initialize-before-use bytecode verification that keeps track of pointer aliasing by keeping track of the line number at which an object was first created. When a pointer to an uninitialized object is copied, the alias analysis algorithm copies the line number where the object was created, so that both pointers can be recognized as aliases for a single object. Of course, if an instruction creating a new object is inside a loop, then there may be many different uninitialized objects created at the same line number. However, the bytecode verifier does not need to work for this case, since Java compilers do not generate code like this. We won't consider any cases with conditional branches in this problem.

Let's examine the contents of stack and any associated line numbers for alias detection after execution of each of the bytecode instructions from above Note that we draw the stack growing downwards in this diagram:

After line:	Stack	line # where created	initialized
1	`Point p`	1	no
2	`Point p`	1	no
	alias for `p`	1	no
3	`Point p`	1	no
	alias for `p`	1	no
	int 3	3	yes
4	`Point p`	1	yes

By using line numbers as object identifiers associated with each pointer on the stack, we keep track of the fact that both pointers on the stack point to the same place after line 2. So when the constructor invoked on line 4 initializes the alias for `p`, we recognize that `p` is initialized as well. Thus the `Print()` method is applied to a properly initialized `Point` object, and this example code fragment passes our simple initialize-before-use verifier.

(a) Consider the following bytecode:

```
1: new #1 <Class Point>
2: new #1 <Class Point>
3: dup
4: iconst 3
5: invokespecial #4 <Method Point(int)>
6: invokevirtual #5 <Method void Print()>
```

When line 5 is reached, there will be more than one uninitialized `Point` objects on the stack. Use the static verification method described above to figure out which one gets initialized, and whether the subsequent invocation of the `Print()` method occurs on an initialized `Point`.

You should draw the state of the stack after each instruction, using the original line number associated with any pointer to detect aliases.

(b) So far we have only considered bytecode operations that use the operand stack. For this problem we introduce two more bytecode instructions, `load` and `store`, which allow values to be moved between the operand stack and local variables.

`store x` removes a variable from the top of the stack and stores it into local variable `x`.

`load x` loads the value from local variable `x` and places it on the top of the stack.

In the following table, we have begun applying the initialize-before-use verification procedure described above to the code fragment in the left column. Note that we maintaining information about aliasing and initialization state for local variable 0 as well as the contents of the operand stack.

Code	local variable 0		stack	
	line #	init?	line #	init?
`1: new #1 <Class Point>`	n/a	no	1	no
`2: new #1 <Class Point>`	n/a	no	1 2	no no
`3: store 0`	2	no	1	no
`4: load 0`				
`5: load 0`				
`6: iconst 5`				
`7: invokespecial #4 <Method Point(int)>`				
`8: invokevirtual #5 <Method void Print()>`				
`9: invokevirtual #5 <Method void Print()>`				

Continue applying the procedure to fill in the missing information on lines 4-8. Based on the state information you have after instruction 7, is the `Print()` method on line 8 applied to a properly initialized object? What about the `Print()` method on line 9?

6. .... Java Generics

For this problem, we will use the following code:

```
class Boat{
  public Boat(boolean paint){
    paint_ = paint;
  }
  public boolean needPaint(){
    return paint_;
  }
...
}
class Sailboat extends Boat{
...
}
class Motorboat extends Boat{
...
}
public class BoatList<E extends Boat>{
  LinkedList<E> list_;

  public BoatList(){
    list_ = new LinkedList<E>();
  }
  public boolean add(E o){
    return list_.add(o);
  }
  public E getFirst(){
    return list_.getFirst();
  }
  public E removeFirst(){
    return list_.removeFirst();
  }
  public int size(){
    return list_.size();
  }
}
```

To answer this question, you should know a little about the *wildcard* type. The wildcard type can be used to define bounds on type variables. Take the following code for example:

```
LinkedList<A> aList = new LinkedList<A>();
LinkedList<B> bList = new LinkedList<B>();
aList.add(new A());
bList.add(new B());
LinkedList<? extends A> wildList = aList;

A a = wildList.getFirst();
wildList = bList;
B b = wildList.getFirst();
```

aList and bList are defined as LinkedLists containing elements of type A and B, respectively. The notation "? extends A" means an unknown type that is a subtype of A. Thus, wildList is defined as a LinkedList of elements of any subtype of A, including A. Because aList points to a LinkedList$\langle$A$\rangle$, the assignment of aList to wildList does not produce a compile-time type error. Since bList points to a LinkedList$\langle$B$\rangle$ and B is a subtype of A, the assignment of bList to wildList also does not produce an error. The compiler knows that wildList's elements are subtypes of A, so it allows the assignment of wildList.getFirst() to a variable a of type A. However, the assignment of wildList.getFirst() to variable b of type B is not permitted. This is because wildList could hold elements of any subtype of A, including elements of type A. Since it would be a type error to assign an element of type A to a variable of type B, "b = wildList.getFirst()" results in a compile-time type error.

For a more in-depth discussion of Java generics, check out Sun's generics tutorial at http://java.sun.com/j2se/1.
tutorial.pdf.

(a) We want to design a static function that takes a BoatList and returns the number of boats in the BoatList that need paint. Fill in the following definition of `countNeedPaint()` by replacing the ellipses ... with Java code.

```
public static int countNeedPaint(...){
  int count = 0;

  while (bl.size() > 0){
    Boat b = ...
    if (b.needPaint()){
      count++;
    }
  }
  return count;
}
```

(b) Next, we want to define a static function that returns the first element of a BoatList. Here are two possible definitions of this function:

```
public static <G extends Boat > G getFirst(BoatList<G> bl){
  return bl.getFirst();
}

public static Boat getFirst(BoatList<? extends Boat> bl){
  return bl.getFirst();
}
```

Would either of these functions cause the Java compiler to generate a compile-time error? Which definition of this function should you choose? Explain briefly.

(c) Finally, we want to design a static function that adds an element to a BoatList. Fill in the following code snippet. (Hint: this function should be able to add a Boat to a BoatList⟨Boat⟩, a Sailboat to a BoatList⟨Boat⟩, and a Sailboat to a BoatList⟨Sailboat⟩.)

```
public static <...>
       void addElement(...){

  list.add(...);
}
```