Homework 8

Due 03 December

Handout 23
CS242: Autumn 2002
26 November

Reading

1. Chapters 11 and 12 in the course reader.

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,
Meat, for pizza that has meat topping,
Veg, for pizza that has vegetable topping,
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.

For simplicity, treat sausage and ham as meats and pineapple and mushroom as vegetables.

(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

As discussed in this chapter, 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.

```
public swapper (Object[] swappee){
   if (swappee.length > 1){
     Object temp = swappee[0];
     swappee[0] = swappee[1];
     swappee[1] = temp;
   }
}
```

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 the principle 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 may insert run-time checks to determine that all the objects are of the correct type. What run-time checks are inserted 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 all right 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. .... Java Bytecode Analysis

One property of a Java program that is checked by the verifier is that each object must be properly initialized before it is used. This property is fairly difficult to check. One relatively simple part of the analysis, however, is to guarantee that each subclass constructor must call the superclass constructor. The reason for this check is to guarantee that the inherited parts of every object will be initialized properly. If we were designing our own bytecode verifier, there are two ways we might consider designing this check:

(i) The verifier can analyze the bytecode program to make sure that on every execution of a subclass constructor, there is some call to a superclass constructor.

(ii) The verifier can check that the first few bytecode instructions of a subclass constructor contain a call to the superclass constructor, before any loop or jump inside the subclass constructor.

In design (i), the verifier should accept every bytecode program that satisfies this condition, and reject every bytecode program that allows some subclass constructor to complete without calling the superclass constructor. In design (ii), some subclass constructors that would be acceptable according to condition (i) will be rejected by the bytecode verifier. However, it may be possible to design the Java source code compiler so that every correct Java source code program is compiled to bytecode that meets the condition described in design (ii) above.

(a) If you were writing a Java compiler, and another person on your team were writing the bytecode verifier, which design would you prefer? Explain briefly.

(b) If you were writing a Java compiler and your manager told you that the standard verifier used design (ii) instead of (i), could you still write a decent compiler? Explain briefly.

(c) If you were writing a bytecode verifier, and your manager offered to double your salary if you satisfied design condition (i) instead of (ii), but fire you if you fail, would you accept the offer? Explain in one sentence.

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		*Closures*	*Compiled Code*
(1)	`Principal`	`SYSTEM`		
(2)	`Principal`	`UNTRUSTED`		
(3)	control link	(2)		
	access link	(1)		
	`url.open`	•	⟨( ), • ⟩	
(4)	control link	(3)		code for `url.open`
	access link	(3)		
	`file.open`	•	⟨( ), • ⟩	
(5)	control link	(4)		code for `file.open`
	access link	(2)		
	`foo`	•	⟨( ), • ⟩	
(6) `sys.main`	control link	(5)		code for `foo`
	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.