

# Programming Languages Assignment 4 Solutions

## 8.4: Look at 8.1 instead, we did this in class.

### 13.2

- (a) One situation where you would want to mark a class as final is when you do not want clients of the class to be able to create new subtypes of it. For example, the String class in the Java libraries is final since that class is part of the definition of Java and allowing people to create subtypes of String would have serious security consequences. A situation where you may want only one method to be final is when you would like to be able to create subclasses of a class, but still guarantee that certain operations always work in the same way on all subclasses. For example, the Process class in the Java libraries is not final, so we can create our own kinds of processes, but methods like stop are final since we always want stop to be able to same as the stop defined in Process. Otherwise, someone could create an unstoppable process.
- (b) Both the Java final construct and the C++ use of non-virtual statically guarantee which code will be used when a certain method is invoked. However, in Java, a method can be non-final (*i.e.*, virtual) in the base class and final in the derived class. In C++, once a method is made virtual, it can never be non-virtual in a derived class. Since final methods might have been virtual in the base class, they must be put into the method lookup table; in C++, non-virtual methods are not put into the vtable. All Java methods are dynamically looked up in the method lookup table at runtime, so even though we can sometimes statically determine that a final method has been called, we must still look it up at runtime. In C++, on the other hand, non-virtual methods do not require a run-time method lookup.
- (c) The Java finalize method is useful since it allows an object to release any resources it holds before being garbage collected. For example, any open files held by an object could be closed in the finalize method.

### 13.4

- (a) Normal tree except Hawaiian inherits from Ham and Pineapple
- (b) Some Hawaiian members may be inherited from Meat and Veget. Pizza fields may be duplicated, or they may be conflicts between Meat and Veget. definitions.
- (c) Anything reasonable, only leaves can be classes to make things work properly. You'd define both interfaces and implementations for Ham, Sausage, Pepperoni, etc.
- (d) C++ multiple inheritance lets you inherit implementation from multiple base classes, so it is easier to reuse code. However, there are some anomalies associated with multiple inheritance, particularly when the base classes have one or more members in common. The Java language design is simpler than C++ in this particular regard.

### 13.7

- (a) It is a standard part of all exception mechanisms to deallocate activation records when an exception is thrown. This deallocates any objects that are on the stack. The extra design decision in C++ is to call the destructor on each object that is deallocated along with the activation record that contains it. It is useful to call each destructor since objects may either hold locks on other resources, or hold pointers to other objects that need to be deallocated. For example, suppose the first cell of a linked list is on the stack. Then the destructor for this object on the stack can call the destructors of all the objects on the list. This cleans up the whole list, instead of just the object on the stack. If the destructor were not called, there would be a memory leak.
- (b) It would not be correct to call the finalize method of every object reachable from the stack. The reason is that some objects reachable from the stack may also be reachable from the program after the exception is processed. The finalize methods of these objects should not be called until these objects are unreachable and they are garbage collected. (It might not be a bad idea for the garbage collector to be called after an exception is caught, however.)

### 12.1

- (d)
- (i) The way assignment of stack objects works can be deduced from: A. The size of activation records is fixed at compile time.
  - (ii) We want to maintain type safety. From (a) we know that `d2`'s `y` cannot be copied into `b2`, so we just copy `x`. The question that remains is which vtable pointer to use. Functions defined in `MiniVan` might access field `y`. Since `b2` does not have space for `y`, `b2` must remain a `Vehicle` object. Otherwise a type error could occur when we call `MiniVan`'s `f` on `b2`. Therefore `b2`'s vtbl pointer points to the `Vehicle` vtable.
- (e) For each assignment or binding of a value to a variable, we have to check if the type of the value is a subtype of the type of the variable. `d1 = b1`: The type of the value (`b1`) is `Vehicle*`, which is *not* a subtype of the type of the variable (`d1`), `MiniVan*`. `d2 = b2`: The type of the value (`b2`) is `Vehicle`, which is *not* a subtype of the type of the variable (`d2`), `MiniVan`. We do not want to bind a variable to a value which is not a subtype, because future operations on that variable may not work, since they are not available on the value.