CS558 Programming Languages – Fall 2023 – Suggested Study Question Solutions for Lecture 10a

1. (a)

(i) This code will cause a static typechecking error at the definition of class `Lion`, because it is declared to be a subclass of abstract class `Animal` but it does not define an `eat()` method.

(ii) This code is valid. It is fine for a subclass like `Lion` to define methods in addition to those specified in the `Animal` superclass.

(iii) This code will cause a static typechecking error in the definition of list `animals`, because `Lion` is not explicitly declared to be a subclass of `Animal`, so it cannot participate in a list of `Animal` even though it actually declares the two methods required for that class.

(b)

(i) This code will not cause an error, but the invocation of `eat()` on the `Lion` object will run the method in the superclass and return the `default eat` string, which may not be what the programmer intended.

(ii) This code runs without error.

(iii) This code runs without error. This contrasts with the behavior of the statically-typed language, which conservatively gives a static error here. Thus the dynamically-typed language is more flexible, which may be useful (although since the programmer did not declare `Lion` to be an `Animal` perhaps `eat()` does not mean what we think!)

(iv) This code will cause a runtime error; the invocation of `eat()` on the `Lion` object cannot be resolved to a method at all. For example, in the Smalltalk language, the analogous code will generate the checked runtime error "message not understood." Here the lack of compile-time checking is a loss; this is in some sense the flip side of the previous example.

2. The program prints 1 2 2 and then throws an uncaught `ClassCastException`.

This example explores how method inheritance works in Java. Class `Q2` is declared to be a subclass of `Q1`. It *inherits* the `g` method from `Q1`, and *overrides* the `f` method. There is nothing Java-specific about this behavior; it would work the same way in almost any class-based OO language, including dynamically-typed ones like Smalltalk.

The main complexity in Java (and other statically typed languages, such as Scala) is due to the interaction of dynamic dispatch with static typing. The definition and use of `a` are straightforward: we create an object of runtime type `Q1` (i.e., using the constructor call `new Q1()`), store the result in a variable of the same type, and then invoke the `g` method on that object, which in turn invokes the `f` method on that object, which returns `1`.

The story with `b` is a little more complicated. We create an object of runtime type `Q2` and store it in variable of the same type, and then invoke the `g` method on that object. This works because `Q2` inherits the definition of `g` from `Q1`. Perhaps surprisingly, `g`'s call to `f` invokes the version of `f` in `Q2`, not in `Q1`, despite the fact that the code for `g` is syntactically contained in `Q1`. This is because all method calls are resolved by considering the *runtime type of the receiving object*. (One language in which this is *not* the default method of resolving calls is C++, in which only methods declared as `virtual` use runtime dispatch; by default, method dispatch uses the *static* type of the receiving object to determine which method version to invoke.)

The behavior of object `c` clarifies that it is the *runtime* type of the reciever that matters. Again, we create an object with runtime type `Q2`, but here we assign it to a variable of type `Q1`. This is legal because a subclass is also a subtype; thus, any object whose runtime type is the subclass can be used wherever a value of the superclass is needed. Even though `c`'s static type is `Q1`, the behavior of the call to `g` is just as in the previous case: we get the version of `f` in `Q2`, not the one in `Q1`, because `c`'s runtime type is `Q2`.

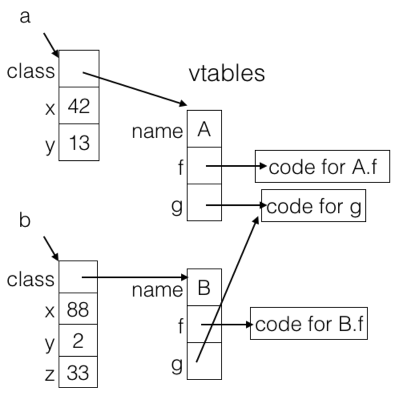
Finally, `d` demonstrates that the subtyping relation is not symmetric. Here we create a `Q1` object and try to assign it into a variable of type `Q2`. The checked cast operation `(Q2)` is needed to make the code compile; if we omit it we will get a compiler error message because `Q1` is *not* a subtype of `Q2`. But the cast only delays the error: when Java tries to cast the `Q1` object to `Q2` at runtime, it sees that this is impossible and raises an exception.

3. (a) The call `b.f()` invokes the function `f(a)` defined within the body of class `A`, passing a `B` object as the value of `a`. The low-level code generated for `f` doesn't know whether it is being invoked on an ordinary `A` object or on an object of some subclass such as `B`; it expects to find the `x` and `y` fields of `a` in the same slots regardless.

(b)

```
A.f(a) = a->x + y->y
B.f(b) = b->x + b->z
g(a) = a->class->f(a)
w = a->class->g(a) + b->class->g(b)
```

Here is the layout:



The object records do not change, but the vtables for classes `A` and `B` do. Both now have an entry for `g` pointing to the same code (which `A` inherits from `B`. But `B` now has a different entry in the slot for `f` than `A` does. The semantics on slide 14 dictate that the call from `b.g()` to `f` should be dynamically dispatched via the vtable slot for `b` to `B.f`, rather than being statically resolved to `A.f`.

(c) Object `c` can be used as a member of class `A` or class `B`, and hence both `f` and `g` can be invoked on it. The corresponding low-level code is:

```
f(a) = a->x
g(b) = b->y
v = c->class->f(c) + c->class->g(c)
```

But `f` expects to find `x` in the first slot of its object argument whereas `g` expects to find `y` there, and there is no way they can both be satisfied, no matter how we lay out `C`. In other words, the problem is that `x` and `y` cannot *both* be the initial prefix of the field order for class `C`.