C M S I   3 8 6

Quiz 2 Answers

1. (a) Under static scope, the script prints 1 * 1 - 1 = 0, (b) Under dynamic scope, the script prints 1 * 3 - 1 = 2.

2. There's something about scope here, because `Failure` is a member class, and an instance of this class is thrown outside of the scope of the name `Failure`. So we can't say `catch (Failure)` to ever catch an exception of this class:

```
// THIS WILL NOT COMPILE
try {
    fail();
} catch (Failure e) {
    // ...
}
```

   since the name `Failure` is not visible there. Or, oh, whoa, maybe it could be in scope there, but it would be in the scope of a *different* binding! Weird. However, we *can* say this:

```
try {
    fail();
} catch (Exception e) {
    System.out.println(e.getClass().getName());
}
```

   and we'll get something like `MyApplication$1Failure`. If necessary we can dig into the result of `e.getClass()` to get at any fields or methods (if there were any), but only by reflection, and never in straight Java code.

3. Lots of languages love to manage their call stacks as actual stacks, and pop off "activation records" when a function ends. But in this case, when *f* returns, we can't really pop its activation off the stack, because its x and y are needed in the returned function. This ruins the stack behavior of calls and requires some smart mechanism to save it, or an implementation that just deals with fragmentation. This is the cost of closures, and a cost that many language designers are not willing to impose on their implementors and users.

4. Yeah it's fine in Python; you either get 3 for static or 5 for dynamic.

   The Ruby version of this problem is much harder, so I've left the answer to that problem here from last year's exam.

   That Ruby code raises a `NameError`, with the message "undefined local variable or method `x' for main:Object", because in the body of *f*, there is no *x* defined. To do the test properly, we have to set things up so there are two variables called *x*, one defined globally and one defined local to *g*. The way you do this is:

```
$x = 3
def f(); puts $x; end
def g(); x = 5; f(); end
g()
```

This prints 3. There is simply no way in Ruby to be inside of *f* and refer to the *x* defined in *g*.

You might also try to get this code fragment in a class, but easier said than done:

```
# RUNTIME ERROR
class C
  x = 3                       # what does this mean?  Are you sure?
  def f(); puts x; end
  def g(); x = 5; f(); end
  g()                         # Again, huh?
end
```

so you you try making a field

```
class C
  def initialize; @x = 3; end
  def f(); puts @x; end
  def g(); x = 5; f(); end
end
C.new.g
```

but of course this still prints 3, just like before. One last try: nest everything inside a function.

```
def try_again()
  x = 3
  def f(); puts x; end
  def g(); x = 5; f(); end
  g()
end
try_again()
```

but that did not allow the *x* to be seen in *f* either. Bottom line is that the static vs. dynamic question can't be answered with a test like this.

Further reading here.

Now what you start to realize is that Ruby's protecting its variables pretty well, here, so how about making them methods?

```
def x; 3; end
def f; puts x; end
def g; def x; 5; end; f; end
g
```

Wow! That printed 5! Dynamically scoped, then? Well let's see. Call x now, again, in the outer scope. Ah it's 5. So the execution of `def x` in *g* overwrote the old binding. There's only one *x* method in the global object.

5. The compiler can always allocate space for a pointer because it knows how big a pointer is. But there is no such thing as an instance of an abstract class. We can only have instances of concrete subclasses. The compiler has no idea what to allocate or how big such a thing would be, since, it, well, doesn't exist.