Solution:

1:
A: This sends the message swim to the object bound to octopus with the argument "fast" (a string)

B: Same as A - parens optional for one-argument methods.

C:This sends the message tentacles= to the object bound to octopus with the argument 8

D: This creates a new instance of the class Aquarium with the argument “clownfish” sent to the initialize method of the newly-created instance.

E: This sends the each message to a 2-element array with the contents "clown" and "fish". The each message takes a block (in curly brackets). The result is that each string is printed.

F: This sends the map message to the 3-element array with the elements 1,2,3. The map message takes a block, and returns a new array with the results from evaluating the block for each element. The result is [10,20,30]

G:The number 4 (an instance of Fixnum) gets the message times with the given block. The block is evaluated 4 times, so that sum becomes 40.

2:

```
class Book
      def initialize(title, author)
            @title=title
            @author=author
      end

      def bookinfo
            yield(@title, @author)
      end

      attr_reader :title, :author
end

bk.bookinfo {|title, author| puts “title is “ + title + “, author is “ author}
```

Bookinfo_closure changes: Add a `pr` parameter next to **bookinfo_closure** that represents

taking in a proc; change `yield` to `pr.call`

3:

```
class Delay
  def initialize &exp
    @exp=exp
    @val=nil
    @evaled=false
  end

  def force
    if !@evaled
      @val=@exp.call
      @evaled=true
    end
    @val
  end
end
```

4:

```
   a) Fine
   b) Error (beverage does not have sugar)
   c) Error (beer does not have method more_expensive)
   d) Fine (sugar is declared implicitly, with value nil)
   e) Error (sugar has value nil which can't be compared to integers)
   f) Fine
   g) Fine (now sugar is no longer nil)
```

5. The following deal with the `MyList` linked-list class, which you should download/copy into a text file from the "blocks_inheritance" file from the course webpage.

a. Write a `filter_block` method that acts like Haskell's filter function, using blocks. In other words, write a method that takes in a block and returns a new `MyList` object with all the elements of the self object that return true when passed into the block.

Solution:

```
def filter_block
  if @tail.nil?
    if yield @head
      MyList.new(@head, nil)
    else
```

```
      nil
    end
  else
    if yield @head
      MyList.new(@head, @tail.filter_block {|x| yield x})
    else
      @tail.filter_block {|x| yield x}
    end
  end
end
```

b. Write a `filter_proc` method that acts like filter_block, but explicitly takes in a proc instead of a block.

Solution:

```
def filter_proc pr
  if @tail.nil?
    if pr.call @head
      MyList.new(@head, nil)
    else
      nil
    end
  else
    if pr.call(@head)
      MyList.new(@head, @tail.filter_proc(pr))
    else
      @tail.filter_proc pr
    end
  end
end
```

c. Write some code to test each of the above methods in the same file. Don't write actual unit tests; instead, write a to_s method for `MyList` and then use `puts` to print the result of multiple tests when running the file. (In other words, this is just practice for writing/using procs and blocks in ruby.)

Solution:
Here are some tests we could run to confirm our solutions. These are not necessarily comprehensive, but demonstrate how we can practice using blocks and procs:

```
def test_filter_block
  list = MyList.new(5, MyList.new("Hi", MyList.new(6.0, nil)))
  puts list.filter_block {|x| x.nil?}
  #assert_equal(list.filter_block {|x| x.nil?}, list)
end
def test_filter_proc
  list = MyList.new(5, MyList.new("Hi", MyList.new(6.0,nil)))
  bproc = Proc.new {|x| if x.nil? then false else true end}
  puts list.filter_proc bproc
  #assert_equal(list.filter_proc(blocc), list)
  list2 = MyList.new(-1, MyList.new(2, nil))
  aproc = lambda {|x| x > 0}
  puts list2.filter_proc aproc
  #assert_equal(list.filter_proc aproc, MyList.new(2, nil))
end

test_filter_block
test_filter_proc
```

6. This will again use the blocks_inheritance file, but will involve the `Point` classes. Define a `PointCircle` class that represents a circle using a point object that represents a circle whose center is at the origin and which takes in a point which lies somewhere on the edge of the circle (in other words, the radius of the circle is the distance from the origin to the point). Clients should be able to access the radius of the circle with a method in `PointCircle`. What kind of Object do you need to pass into `PointCircle`? Is `PointCircle` a `Point`?

Solution:

```
class PointCircle
  Attr_reader :p1
  def initialize(p1)
    @p1 = p1
  end
  def radius
```

```
    p1.distFromOrigin
  end
end
```

This class can take in anything that has a distFromOrigin method. In other words, it can take in any kind of `Point`, as well as any other object with such a method. From a duck-typing standpoint, we can be just as general with the following:

```
def radius
  Math.sqrt(p1.x*p1.x + p1.y*p1.y)
end
```

This class would be able to take in anything that has an x method whose value has a * method that can be sent the result of the point's x method as well as a + method, and a y method whose value has a * method that can be sent the result of the point's y method, whose result can then be sent to the aforementioned + method.

Aside from the syntactic differences between the two solutions, the latter will return a different distance for a ThreeDPoint than the former.

It's important to realize that, although we could make `PointCircle` extend the `Point` class, but it seems more appropriate not to consider `PointCircle` a `Point`. This could be more clear if we added more methods to `PointCircle`, like diameter or circumference. Doing that, we'd be able to see that `PointCircle` and `Point` are describing fundamentally different things.