1. For each of the questions below, suppose the following is invoked:

```
1  B b = new B();
2  b.f();
```

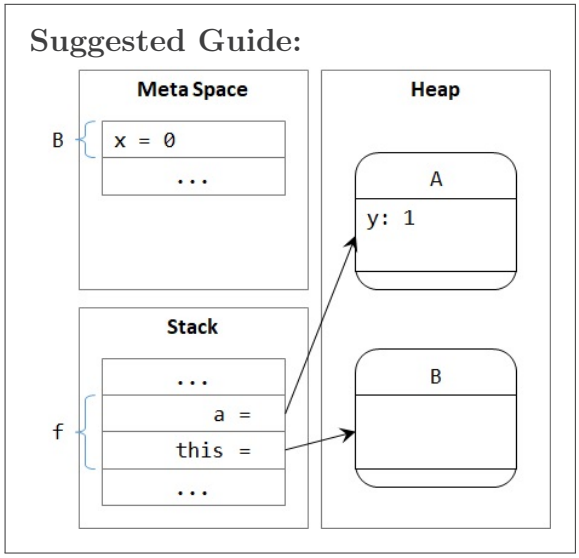
Sketch the content of the stack, heap, and metaspace *immediately after* the line

```
1  A a = new A();
```

is executed. Label the values and variables/fields clearly. You may assume that `b` is already on the heap and you can ignore all other content of the stack and the heap before `b.f()` is invoked.

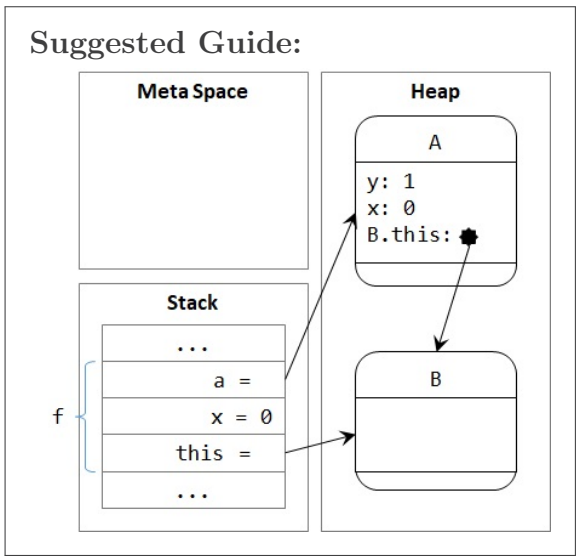
(a) Problem #A

```
1   class B {
2     static
        int x = 0;
3
4     void f() {
5       A a = new A();
6     }
7
8     static class A {
9       int y = 0;
10
11      A() {
12        y = x + 1;
13      }
14    }
15  }
```



(b) Problem #B

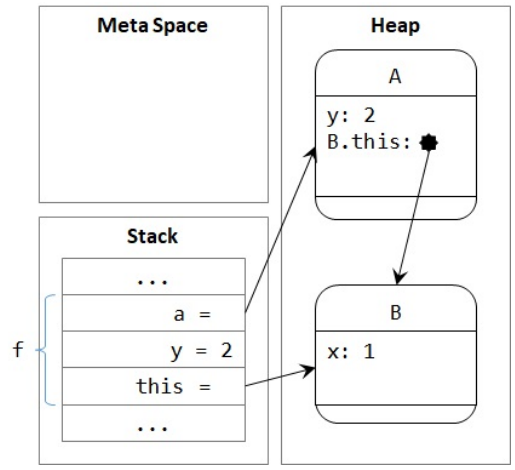
```
1   class B {
2     void f() {
3       int x = 0;
4
5       class A {
6         int y = 0;
7
8         A() {
9           y = x + 1;
10        }
11      }
12
13      A a = new A();
14    }
15  }
```



(c) Problem #C

```
1  class B {
2    int x = 1;
3
4    void f() {
5      int y = 2;
6
7      class A {
8        void g() {
9          x = y;
10       }
11     }
12
13     A a = new A();
14     a.g();
15   }
16 }
```

Suggested Guide:



This is before `a.g()`. *After that, the value of* `x` *in the instance pointed to by* `b` *will be* `2` *instead.*

2. Consider the following `Stack` implementation on the next page. Try running the following code.

```
1  Stack<Integer> s = Stack.getEmptyStack();
2  s.push(1);
3  s.push(2);
4  s.push(3);
5  s.head();
6  s.pop();
7  s.head();
8  s.pop();
9  s.head();
10 s.pop();
```

Change the implementation of `Stack` to make it immutable and create a new class `ImmutableStack`.

```java
public class Stack<T> {
  private T head;
  private Stack<T> tail;
  private static Stack<?> EMPTYSTACK = new Stack<>(null, null);

  private Stack(T head, Stack<T> tail){
    this.head = head;
    this.tail = tail;
  }

  public void push(T t){
    this.tail = new Stack<T>(this.head, this.tail);
    this.head = t;
  }

  public void pop(){
    if (this.head == null) {
      throw new RuntimeException("Stack is empty");
    }
    this.head = this.tail.head;
    this.tail = this.tail.tail;
  }

  public T head(){
    if (this.head == null) {
      throw new RuntimeException("Stack is empty");
    }
    return head;
  }

  public boolean isEmpty(){
    if (this.head == null) {
      return true;
    } else {
      return false;
    }
  }

  public static <T> Stack<T> getEmptyStack(){
    @SuppressWarnings("unchecked")
    Stack<T> emptyStack = (Stack<T>) EMPTYSTACK;
    return emptyStack;
  }
}
```

Suggested Guide:

```
1  public class ImmutableStack<T> {
2    private final T head;
3    private final ImmutableStack<T> tail;
4    private final static ImmutableStack<?> EMPTYSTACK =
5          new ImmutableStack<>(null, null);
6
7    private ImmutableStack(T head, ImmutableStack<T> tail){
8      this.head = head;
9      this.tail = tail;
10   }
11
12   public final ImmutableStack<T> push(T t){
13     return new ImmutableStack<T>(t, this);
14   }
15
16   public final ImmutableStack<T> pop(){
17     if (this.head == null) {
18       throw new RuntimeException("Stack is empty");
19     }
20     return this.tail;
21   }
22
23   public final T head(){
24     if (this.head == null) {
25       throw new RuntimeException("Stack is empty");
26     }
27     return head;
28   }
29
30   public final boolean isEmpty(){
31     if (this.head == null) {
32       return true;
33     } else {
34       return false;
35     }
36   }
37
38   public final static <T> ImmutableStack<T> getEmptyStack(){
39     @SuppressWarnings("unchecked")
40     ImmutableStack<T> emptyStack =
41         (ImmutableStack<T>) EMPTYSTACK;
42     return emptyStack;
43   }
44 }
```

The tester code can then be modified as follows:

```
1 ImmutableStack<Integer> s = ImmutableStack.getEmptyStack();
2 s = s.push(1).push(2).push(3);
3 s.head();
4 s.pop().head();
5 s.pop().pop().head();
6 s.head();
```