CSE 401 - Semantics, Type Checking, & Vtables – Week 7 - Solution

1. Suppose we have the following global scope:

```
class Bar { boolean field; public int method(int i, int j); }
class Foo extends Bar { int val; public boolean whoop(int x); }
```

Now, consider the following hypothetical method definition for `Bar.method`:

```
public int method(int i, int j) {
  int r;
  boolean b;
  Foo o;
  if (this.field) {
    o = this;
    b = o.whoop(i + j);
    r = o.val;
  } else {
    r = i * j + 3;
  }
  return r;
}
```

a. What variables (locals, parameters, etc.) are defined in the *local* scope in the `method` body?

```
Bar this; int i; int j; int r; boolean b; Foo o;
```

Remember that every MiniJava method has an implicit parameter "**`this`**" for the receiver object. For the sake of type-checking the method body, it makes sense to treat it like a normal parameter, although you may treat it however you'd like in your symbol tables.

b. When we execute this method body, a runtime error could result. Explain how something could go wrong by giving values of the parameters and/or variables involved that would cause a runtime error.

```
this = Bar(field: true);
```

The error here is the potential failure of the downcast in the assignment "`o = this`." Unlike real Java, MiniJava's dynamic semantics defines no behavior for a failing downcast, so the static semantics forbids downcasts altogether.

c. The method body also has type errors. Can you describe which type check(s) the compiler could use to deduce this fact?

   Since MiniJava's static semantics forbids downcasts, a MiniJava compiler must check that the type of an assignment statement's right-hand side is either the same as the left-hand side's type or a subclass type of the left-hand side's class type.

d. Does *every* possible execution of this method produce a runtime error? Can you describe any that happen to be statically correct? (Again, possible runtime values for parameters/variables would suffice.)

   No, some possible executions of the method avoid the branch that causes an issue, for example given the following value of **`this`**:

   ```
   this = Bar(field: false);
   ```

   Alternatively, some possible executions could enable the "downcast" to succeed, if the receiver object (**`this`**) ends up really being an instance of the subclass **`Foo`**, like so:

   ```
   this = Foo(field: true, val: <any integer>);
   ```

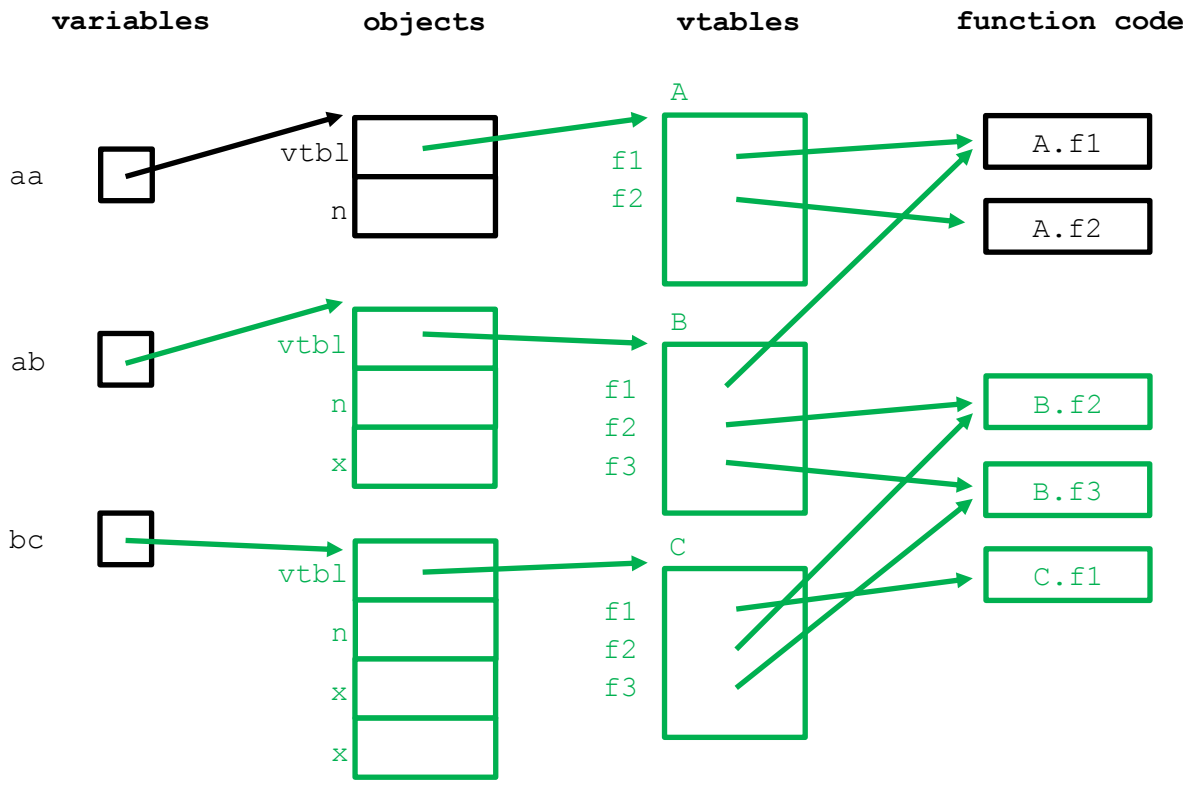
e. Suppose that we replaced the use of **`this`**`.field` in the method body to call a boolean method that always returns false. How would this change your answers to the previous questions?

   Even though the ill-behaving branch would never get run, type checking composes through types and type signatures (*not* the specific values!), so a type checker for MiniJava will not verify the method body (*i.e.*, will report a type error), despite the forbidden behavior being impossible according to the dynamic semantics.

2. Consider the following Java program:

```
class A {
  int n;
  public void f1() { System.out.println("A.f1"); this.f2(); }
  public void f2() { System.out.println("A.f2"); }
}
class B extends A {
  int x;
  public void f3() { System.out.println("B.f3"); this.f1(); }
  public void f2() { System.out.println("B.f2"); x = 11; n = 22; }
}
class C extends B {
  int x;
  public void f1() { System.out.println("C.f1"); this.f2(); x = 33; }
}
class Main {
  public static void main(String[] args) {
    A aa = new A();
    A ab = new B();
    B bc = new C();
  }
}
```

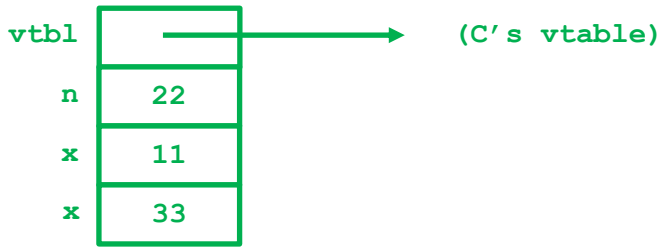
a. Complete the diagram below to show the layout of objects and vtables by the end of the main function:



b. If we added each of the lines below to the end of main, what would the output of the program be? If the line would cause an error, describe why.

`aa.f1();`	**A.f1** **A.f2**
`ab.f1();`	**A.f1** **B.f2** **(Note that even when evaluating an inherited method, "this" refers to the current object)**
`ab.f3();`	**Compiler error – f3 is not defined for class A (even though the underlying object is of type C, the compiler only knows variable types at compile time)**
`bc.f3();`	**B.f3** **C.f1** **B.f2**

c. Suppose we call `bc.f1()`. Draw the bc object after the call, including both its layout in memory and the value stored at each location.



Note that when B.f2 is compiled in B's scope, the variable name "x" refers to B's field x, so the reference to it gets compiled to store a value at the offset of the x field. Later, this method is inherited by C, but that doesn't change it's meaning. Since it was compiled to use the offset of B's field x, it will still use that offset even when executing as an inherited method of C (which happens to also have a field named x). This behavior is why it's important for subclass objects to have space for all the fields of their superclass: if they inherit any methods that reference fields, those methods should still work with the offsets they were compiled with.