Exercise 6

Bytecode verification

1)

- Object `a` behaves like a normal cell. Object `b` is also a cell, but it increases the stored value by 1. The interesting difference is between `c` and `d`. They are both cells. They have mixed in exactly the same traits. However, calling `set(i)` has a different effect on them: it stores `2i+1` to the first one and `2(i+1)` to the second one.
- Trait `Doubling` will not get mixed in twice, as perhaps the programmer would expect. Scala rejects this statically.

  The problem can be bypassed in an ugly way, by creating a new trait `Doubling2` that behaves exactly like `Doubling` and then introducing `e = new Cell with Doubling with Doubling2`. Here is our first try:

  ```
  trait Doubling2 extends Doubling
  val e = new Cell with Doubling with Doubling2
  ```

  The code passes through, but dynamically `e` behaves as if it were a `Cell with Doubling`. Scala lets the code go through, because `Doubling2` may introduce new functionalities, but refuses to include `Doubling` twice in the linearization.

  Our last try, the ugliest of all, but the one which will finally work, is to create a whole new trait from scratch, reusing nothing:

  ```
  trait Doubling3 extends Cell
  {
    override def set(i:int) = { super.set(2*i) }
  }
  val e = new Cell with Doubling with Doubling3
  ```

  And now `e.set` quadruples its argument as expected.
- We can produce the problem using the traits provided, but here is a more interesting case. Consider the following code:

```
class C
{
    def m() = { println("m executing") }
}

trait Logging extends C
{
    val logFileName: String
    override def m() =
    {
          println("Logging to: " + logFileName)
          super.m()
    }
}

class C1 extends
    C with Logging
    { override val logFileName = "A" }
          // this class logs all calls to m
```

```
            // to a file named "A"
```

Suppose now that we give the client the classes `C`, `C1` and the trait `Logging`. The client has no knowledge that `C1` was created using `Logging`. The client wishes to log calls to `m` to a file called "`B`". The client does this for both classes `C`, `C1`.

```
class C2 extends
    C with Logging
    { override val logFileName = "B" }

class C3 extends
    C1 with Logging
    { override val logFileName = "B" }

object LogEx1
{
    def main (args:Array[String]) =
    {
            val a = new C2
            val b = new C3
            a.m
            b.m
    }
}
```

The call `a.m` works as it should: the method call is logged to file "`B`" only.

However, the call `b.m`, does not behave as it should: it logs the call to `m` only to file "`B`", even though it is an instance of `C1`, which is supposed to log calls to `m` to file "`A`" too.

The problem is that, unbeknownst to the client, the trait `Logging` has been mixed in twice. This overrode its initial behaviour, interrupting the logging to "`A`".

2)

- No. The dynamic type of `x` can be mixing in traits that break the invariant. Even if we suppose that the only traits that can extend `Cell` are `Incrementing` and `Doubling`, this is not enough to enforce behavioural subtyping. In particular, an object of type `Cell with Incrementing with Doubling` can be still passed as argument to method `foo` in this restricted context, and this would break the invariant.
- Consider the following example:

```
class C
{
        var x:int;
        def foo() = {} //ensures true
}
trait T1 extends C
{
        override foo() = { x=x+1 } //ensures x>old(x)
}
trait T2 extends C
{
        override foo() = { x=x-1 } //ensures x<old(x)
}
```

Both `C with T1` and `C with T2` are behavioral subtypes of `C`. But `C with T1 with T2` is not a subtype of `C with T1`.

3)

- Here ([], [E,b,b,C1,C2,A]) is initial state. We denote the type boolean as b for convenience (in reality the Java bytecode verifier views it as an integer).

0	`iload_1`	([b], [E,b,b,C1,C2,A])	[b], [E,b,b,B,A,A])	([b], [E,b,b,A,A,A])
1	`ifeq 22`	([], [E,b,b,C1,C2,A])	([], [E,b,b,B,A,A])	([], [E,b,b,A,A,A])
4	`iload_2`	([b], [E,b,b,C1,C2,A])	([b], [E,b,b,B,A,A])	([b], [E,b,b,A,A,A])
5	`ifeq 12`	([], [E,b,b,C1,C2,A])	([], [E,b,b,B,A,A])	([], [E,b,b,A,A,A])
8	`aload_3`	([C1], [E,b,b,C1,C2,A])	([B], [E,b,b,B,A,A])	([A], [E,b,b,A,A,A])
9	`goto 14`	([C1], [E,b,b,C1,C2,A])	([B], [E,b,b,B,A,A])	([A], [E,b,b,A,A,A])
12	`aload 4`	([C2], [E,b,b,C1,C2,A])	([A], [E,b,b,B,A,A])	([A], [E,b,b,A,A,A])
14	`astore_3`	([B], [E,b,b,C1,C2,A]) → ([], [E,b,b,B,C2,A])	([A], [E,b,b,B,A,A]) → ([], [E,b,b,A,A,A])	([A], [E,b,b,A,A,A]) →([], [E,b,b,A,A,A])
15	`aload 5`	([A], [E,b,b,B,C2,A])	([A], [E,b,b,A,A,A])	([A], [E,b,b,A,A,A])
17	`astore 4`	([], [E,b,b,B,A,A])	([], [E,b,b,A,A,A])	([], [E,b,b,A,A,A])
19	`goto 0`	([], [E,b,b,B,A,A])	([], [E,b,b,A,A,A])	([], [E,b,b,A,A,A])
22	`aload_3`	([C1], [E,b,b,C1,C2,A])	([B], [E,b,b,B,A,A])	([A], [E,b,b,A,A,A])
23	`areturn`	([], [E,b,b,C1,C2,A])	([], [E,b,b,B,A,A])	([], [E,b,b,A,A,A])

In the provided table, each cell contains the output value of a corresponding instruction. Different columns correspond to different iterations. There are two values for the instruction at address 14. The first one is the output of the join operation, and the second one is the output of the corresponding instruction.

- Here the essential information is marked with bold font:

0	`iload_1`	**([],[E,b,b,A,A,A])** →([b], E,b,b,A,A,A])
1	`ifeq 22`	([], [E,b,b,A,A,A])
4	`iload_2`	([b], [E,b,b,A,A,A])
5	`ifeq 12`	([], [E,b,b,A,A,A])
8	`aload_3`	([A], [E,b,b,A,A,A])
9	`goto 14`	([A], [E,b,b,A,A,A])
12	`aload 4`	([A], [E,b,b,A,A,A])
14	`astore_3`	**([A], [E,b,b,A,A,A])** →([], [E,b,b,A,A,A])
15	`aload 5`	([A], [E,b,b,A,A,A])
17	`astore 4`	([], [E,b,b,A,A,A])
19	`goto 0`	([], [E,b,b,A,A,A])
22	`aload_3`	([A], [E,b,b,A,A,A])
23	`areturn`	([], [E,b,b,A,A,A])

4)

- Because the inference algorithm doesn't take interfaces into consideration, the calculated type for the variable `iface` is `Object`.
- Because the inferred type of the `iface` is `Object` the decision can be made only during the execution.
- In both cases the inferred type of the `iface` is `IFace`. The decision about the safety of the call can be made during bytecode verification.

5)

-
  ```
  0 : aload_0
  1 : iconst_1
  2 : ifne 4
  3 : aload_0
  4 : astore_1
  ```

  Note: `ifne` jumps to the given index if the integer value at the top of the stack is not equal to zero. It pops the value at the top of the stack.

  There are two possibilities for the stack size after executing this program. On the other hand, the height of the stack at point 4 is at least 1, and there will be surely a reference value at the top of the stack.
- Yes we can construct such an algorithm. The update is as follows: when joining stacks of different sizes, pick the smallest one, but carry as extra information the size of the largest one to be used when checking for overflow.
  Note that if we just picked the smaller one and used that, we would not prevent stack overflows at runtime.
  If we just picked the largest one and made the "extra" values into dummy values by giving them the "top" type, we might not prevent underflows when using instructions such as pop().
- This limitation is not essential. If we have two states {[head1, x], [head2]} where head1 and head2 are stacks of the same size, then we can't access x.