Exercise 6

Bytecode verification

1)

- 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,B,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])

2)

-

0	`iconst_5`	([],[C)
1	`ifeq 4`	([int], [C])
2	`new A`	([], [C]) → ([A], [C])
3	`goto 5`	([A], [C])
4	`new B`	([], [C]) → ([B], [C])
5	`invokevirtual A.foo()`	([C], [C])

Note: in order to obtain the result of statement 5 the type inference algorithm has to compute the smallest common supertype of A and B.

- Because the inference algorithm infers that at statement 5 the type of the value at the top of the stack is `C`, while the `invokespecial` statement requires a reference of type `A`.
- In general, we can add some static analyses in order to discover and remove unreachable code.
  In this particular case, we can to add a simple constant analysis to the verifier in order to infer that at point 1 the value at the top of the operand stack is 5 and so that the following `ifeq` cannot be true.
  Note that, even if this analysis seems quite trivial, it is something that is not yet part of the Java compiler. For instance, the following Java code:

```
public class B {
 int foo() {
        int i=1;
        i=i*2+12;
        if(i<0)
              i=i*2;
        else i=i*4;
        while(i<0)
              i++;
        return i;
 }
}
```

is compiled to:

```
 0:     iconst_1
 1:     istore_1
 2:     iload_1
 3:     iconst_2
 4:     imul
 5:     bipush 12
 7:     iadd
 8:      istore_1
 9:      iload_1
10:      ifge 20 (+10)
13:      iload_1
14:      iconst_2
15:      imul
16:      istore_1
17:      goto 24 (+7)
20:      iload_1
21:      iconst_4
22:      imul
23:      istore_1
24:      iload_1
25:      ifge 34 (+9)
28:      iinc 1 by 1
31:      goto 24 (-7)
34:      iload_1
35:      ireturn
```

- We can anticipate the execution of method `A.foo()` putting it on the first branch of the ifeq statement.

```
0:      iconst_5
1:      ifeq 5
2:      new A
3:      invokevirtual A.foo()
4:      goto 7
5:      new B
6:      pop
7:      …
```

Note: `pop` simply removes the value at the top of the operand stack. We need it to obtain two stacks of the same height at the end of the method.

- There will be the same problem as before, since we have a reference of type `C` on the top of the stack, and class `C` does not define any `foo()` method.
- If we have only dynamic checks, the program will execute normally method foo(). In fact at runtime we can have only a reference of type A at the top of the stack, and class A defines method foo()
- We can modify the type checker considering the set of the possible types instead of taking the smallest common supertype. In this way it will infer that at the top of the stack we can have a reference of type A or B, and in both the cases method foo() is defined on that class.
- The main reason is probably that we have to find out at runtime which method is invoked, and this may cause a slowdown during the execution. In particular inside a loop, it can be particularly efficient to know which method we are invoking, skipping the runtime lookup.
  On the other hand, the original invokevirtual requires a runtime lookup! In fact, because of method overriding, we could have that the runtime type of the value at the top of the operand stack overrides the method defined in the class specified in the invokevirtual

statement.
In the Java virtual machine specification, this procedure is described as follows:

> *Let C be the class of objectref. The actual method to be invoked is selected by the following lookup procedure:*
>
> - *If C contains a declaration for an instance method with the same name and descriptor as the resolved method, and the resolved method is accessible from C, then this is the method to be invoked, and the lookup procedure terminates.*
>
> - *Otherwise, if C has a superclass, this same lookup procedure is performed recursively using the direct superclass of C ; the method to be invoked is the result of the recursive invocation of this lookup procedure.*
>
> - *Otherwise, an `AbstractMethodError` is raised.*

Looking at this description, it seems that the goal of specifying also the class is not:

- to bound the lookup up to the given class, since the lookup procedure goes on looking to the superclass,
- to be sure that this method exists, since it is possible that the method does not exist and the lookup procedure raise an `AbstractMethodError`).

We can guess and discuss several other reasons, for instance:

- to know the return type of the method, and so to infer and check that the following bytecode statements type check,
- to avoid to call methods that have the same signature (that is, the same name and the same parameters) but that belong to completely different classes. This could be important for security reasons – preventing malicious injection of additional classes which intentionally dublicate existing method signatures.

3)

- 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.

4) Here is an example of such a program:
x=true; x=5;
The type of the variable can change in the bytecode but not in the source code.

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 larger 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.