Exercise 2

Types and Subtyping

1. The right answer is 2 (D <: F <: E).

   A <: B holds since both the classes define a method `int x()`. Also B <: A holds for the same reasons. C <: A holds since class C contains all the methods of class A (`int x()`) plus something else (`int y()`) (but A <: C does not hold since C defines method y that is not part of class A).

   F <: E does not hold since method `B foo(C a)` in F does not contain/overwrite method `B foo(A a)` in `E` since we would need contravariant argument type (that is, A <: C), but, as already pointed out, this does not hold.

2. `B=C <: A`
   `D <: E = G <: F`
   No other subtyping relations exist, except the reflexive and transitive closure of the above.

   The type `J` is not a subtype of `L`, even though it might seem that `J` has a larger interface than `L`. That would be the case if we had `read-only` instead of `immutable`.

3. The compiler allows the code to go through although it can't prove that `c` implements `I`. The reason is that there might be a subclass `D` of `C` such that `D` implements `I` and `c` might be an object of `D`. Here Java opts for the flexibility of dynamic type checking. When the code executes a runtime exception is thrown, because `c` does not implement `I` and this is caught by the runtime check.

4. Class A restricts the accessibility of method `x`, since it is `protected` in B and `private` in A. This means that class A allows fewer behaviors than B, so it cannot be subtype of B (that is, it cannot extend B) since this would contradict the substitution principle. On the other hand, class C relax the accessibility level of method `x`, so it allows more behaviors than B, and this is allowed by the Java compiler.

   In general, a class can be subtype of another class if it assigns "weaker" accessibility permissions that the ones of the superclass.
   In Java, there are four different types of access modifiers for fields and methods:
   - public: every class can access the element
   - protected: only subclasses and classes in the same package can access the element
   - default: only classes in the same package can access the element
   - private: only this class can access the element

   In general, we can state that

   `public <: protected <: default <: private`

where a <: b means that the accessibility level a is weaker than b, and that a subclass can relax the accessibility level b with a.

5. "in" parameters – contravariant. "out" parameters covariant. The rest invariant. Notice that the answer depends on whether a type refers to a value that can be read and/or written by the method. This means that "in out" and "ref" behave similarly as far as the present question is concerned.

6. Consider

```
if (x=x) then y:=1 else y:=true
y:=y+1
```

A usual static type system would reject this program, while the program would not cause typing problems. The static type system would reject the following program which would generate a runtime type error:

```
f(x) { return x+1 }
print f(true)
```

7. The code tries to override a non-existing method. The new method has type `ColoredPoint->bool` and the old method has type `Point->bool`. Since C# classes are invariant in the method parameter types, the new method cannot override the old one. This is reasonable, because the requirement that `ColoredPoint` is a subclass of `Point` entails the following substitution principle: every object of `ColoredPoint` should be useable wherever a point is expected. The substitution principle is not respected whenever a `ColoredPoint c` is compared to a `Point p`, as in `c.isEqual(p)`.

Eiffel would allow the overriding due to its covariance policy. This allows the program to compile. It allows `Point` objects to be compared to `Point` objects and `ColoredPoint` objects to `ColoredPoint` objects. However, the unsoundness above will remain. Eiffel will try to catch this statically by forbidding all calls that would potentially compare objects coming from two different classes. This forbids too much. Also, it does not respect the substitution principle of subtyping.

If we removed the `override` keyword, the program would compile. Due to overloading, `ColoredPoint` will be a subtype of `Point`, supporting two *different* methods:

```
boolean isEqual (Point)
boolean isEqual (ColoredPoint)
```

In Java the same thing would happen. However, a Java programmer used to dynamic dispatch will find the following program surprising:

```
void f ()
{
     ColoredPoint p,q;
     p = new ColoredPoint ();
     p.x = 1; p.y = 2; p.color = 3;
     q = new ColoredPoint ();
     q.x = 1; q.y = 2; q.color = 4;
```

```
        boolean b1 = p.isEqual (q); // b1 == false
        boolean b2 = g (p, q); // b2 == true
    }

    boolean g (ColoredPoint pp, Point qq)
    {
        return pp.isEqual (qq); // returns true
    }
```

If we don't want `ColoredPoint` to be a subtype of `Point`, we are free to ignore the comparison between the two. However, a language with only subclassing, like Java or C#, will force us to rewrite all the members that could have been reused (in this example, these are only `x`, `y`, but in general, this may be a huge rewriting). Languages that decouple subtyping from inheritance, like C++ and Eiffel, do not have this problem.