Exercise 2

Types and Subtyping

1. The right answer is 5 (`F <: E`).

   `C <: B` does not hold since `C` does not contain method `void m1()`. For the same reason `C <: A` does not hold.

   The subtyping relations we have between the classes `A, B, C,` and `D` are:
   `D <: B <:A`
   `D <: C`

   `E <: G` does not hold since the argument of method `foo` in `E` (of type `D`) is not a subtype of the one in `G` (of type `C`).
   `G <: F` does not hold since the return type of method `foo` in class `G` (type `B`) is not a subtype of the one in `F` (type `D`).

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

3. Yes, contravariant arrays would require run-time type checks when reading values from the array. We would need run-time type checks when reading from arrays.
   By definition of contravariance, we have that S<:T then T[]<:S[].
   Then `Object[]<:String[]` since `String<:Object`. So we can pass an array of type `Object[]` to a method that requires a `String[]` argument.

```
class C {
  String foo(String[] a) {
    return a[0];
  }
}

void client(C c) {
  Object[] arr = new Object[1];
  arr[0] = new Object();
  String s = c.foo(arr);
}
```

4. Class `A` restricts the accessibility of method `get`, since it is `protected` in `B` and `private` in `A`. This means that class `A` allows fewer behaviors than `B`, so it cannot be a subtype of `B`. On the other hand, class `C` relaxes the accessibility level of method `get`, 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

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. 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.