Exercise 2

Types and Subtyping

1. Investigate the behavior of the following Java code:

```
interface I {};

class C {};

public class E2_1
{
      public static void main (String [] argv)
      {
            C c = new C ();
            I i = (I) c;
      }
}
```

Try to compile it. If it compiles, try to execute it. What happens? Why?

2. Suppose that we have a language with structural typing, contravariant parameter types and covariant return types. Consider the following types:

```
class A { int m(int x){…}; }
class B { int m(int x){…};  int n(int x) {…}; }
class C { int n(int y){…};  int m(int x) {…}; }
class D { C m(A a) {…}; }
class E { C m(B b) {…}; }
class F { A m(B e) {…}; }
class G { B m(C e) {…}; }
class H { G m(D d, E e) {…}; }
class I { F m(E e, D d) {…}; }
class J { A a; }
class K { B b; }
class L { immutable A a; }
```

Find all the subtyping relations among them. Assume that int has no subtype other than itself.

3. Show:

   - A program that is rejected by a statically typed language but is executed without typing errors in a dynamically typed language.

- A program that is rejected by a statically typed language and runs into a type error when executed in a dynamically typed language

4. In C#, methods the following kinds of parameters are supported:

   - "in" parameters. The caller provides an expression that is passed by value to the formal parameter.
   - "out" parameters. The caller provides a variable as actual parameter. An output from the method is returned and written to the actual parameter when the method terminates. The method does not read the initial value of the actual parameter and the actual parameter has no connection to the formal parameter during the execution of the method.
   - "in out". The same as "out" but the method may also read the initial value of the actual parameter.
   - "ref". The parameter is passed by reference. The caller provides a variable that is aliased by the formal parameter during the execution of the method.

   What should be the variance rules for all these kinds of parameters? Why? Refer to the contra-variance rule for method parameters and the co-variance rule for return values to explain your answer. Give examples that would not work, if your rules are not followed.

5. Consider the following C# classes:

```
class Point
{
     public int x, y;

     public Boolean isEqual (Point p)
     {
         return p.x == x && p.y == y;
     }
};

class ColoredPoint : Point
{
     public int color;

     override public Boolean isEqual (ColoredPoint p)
     {
        return p.x == x && p.y == y && p.color == color;
     }
};
```

   The compiler refuses to compile this code. Why? Is this reasonable?

What would happen if we wrote the same example in Eiffel? Is there any problem?

What would happen if we removed the `override` keyword? Is there a problem now? (Note that the `override` keyword is mandatory if we want to override a method in C# and that overloading of methods is permitted.)

What would happen if we wrote the same example in Java?

What would happen if we removed the requirement that `Point` is a subtype of `ColoredPoint`?