Concepts of Object-Oriented Programming
AS 2014

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 2

Types and Subtyping

October 3, 2014

Task 1

Suppose that we have a language with structural subtyping, 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; }
```

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

Task 2

As you have seen in the lectures, arrays are covariant in Java and C#. Because of this, each array update requires a run-time type check. Another approach would have been to adopt contravariant arrays. Does this solution require run-time type checks? If this is the case, explain in which cases you need these run-time type checks and provide an example in which a check would fail.

Task 3

Consider the following Java program:

```
class B {
   protected int get() {...}
}

class A extends B {
   private int get() {...}
}

class C extends B {
   public int get() {...}
}
```

When we compile it, we obtain the following error:

```
get() in A cannot override get() in B; attempting to
assign weaker access privileges; was protected
        private int get() {...}
                    ^
```

Explain why this is the behavior of the Java compiler.

Task 4

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.

Task 5

Suppose that we have a C#-style programming language supporting the following kinds of parameters:

- "`in`" parameters. The caller provides an expression that is passed by value to the formal parameter.
- "`out`" parameters. The caller provides a variable as the 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 these kinds of parameters? Why? Refer to the contravariance rule for method parameters and the covariance rule for return values to explain your answer. Give examples that would not work, if your rules are not followed.

Could "`in out`" and "`ref`" be subtypes of each other? Explain your answer.

Task 6

Consider the following C# classes:

```
class Point {
   public int x, y;
   virtual 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 `ColoredPoint` is a subtype of `Point`?

Task 7

In C++ object aliasing is achieved using pointers and it is possible to have a pointer to a pointer. Here is an example

```
class X {};

class Initializer {
   public:
      void init(X** x) {
         *x = new X();
      }
};

class Value {
   private:
      X* x = nullptr;
   public:
      Value(Initializer* i) {
         i->init(&x); // The initializer object will set the value of x
      }
};
```

How does the substitution principle apply to values of type pointer to pointer? Is it safe to call methods that have the signature of `init` with a value of type pointer to pointer to a subtype/supertype of X? Why?