Weekly exercises, OO-I, 2015

Problem 1

Here is an easy one on type compatibility.

Given the following program fragment in some hypothetical language:

```
type S1 is struct {
      int y;
      int w;
};
type S2 is struct {
      int y;
      int w;
};
type S3 is struct {
      int y;
};
S3 f(S1 p) { ... };
...
S1 a, x;
S2 b;
S3 c;
int d;
...

a = b;      // (1)
x = a;      // (2)
c = f(b);   // (3)
d = f(a);   // (4)
```

a) Under name compatibility, which of the four statements (1) ... (4) are type correct (and which are not).

b) Same question under structural compatibility.

Problem 2

We have the following classes:

```
class Food {...}
class Cheese extends Food {...}
```

Assume that we have the following functions:

```
int f(c Cheese) {...}

int f'(f Food) {...}
```

someFood is a value of type Food, and someCheese is a value of type Cheese. Then we know that

f'(someCheese) can be substituted for f(someCheese)

that is, whenever we have a call 'f(someCheese)' we may just as well call f' with the same someCheese parameter without causing any static type errors: f' can be said to be a subtype of f.

Why cannot f(someFood) be substituted for f'(someFood)? That is why can not f be said to be a subtype of f'? Give an example of class Cheese (that is a more elaborate Cheese than above) and a definition of f that will create a type error.

Problem 3

Exercise 10.2 in Mitchell:

```
enum shape_tag {s_point, s_circle, s_rectangle };
class point {
  shape_tag tag;
  int x;
  int y;
  point (int xval, int yval)
    { x = xval; y = yval; tag = s_point; }
  int x_coord () { return x; }
  int y_coord () { return y; }
  void move (int dx, int dy) { x += dy; y += dy; }
};
class circle {
  shape_tag tag;
  point c;
  int r;
  circle (point center, int radius)
    { c = center; r = radius; tag = s_circle }
  point center () { return c; }
  int radius () { return radius; }
  void move (int dx, int dy) { c.move (dx, dy); }
  void stretch (int dr) { r += dr; }
};
class rectangle {
  shape_tag tag;
  point tl;
  point br;
  rectangle (point topleft, point botright)
    { tl = topleft; br = botright; tag = s_rectangle; }
  point top_left () { return tl; }
  point bot_right () { return br; }
  void move (int dx, int dy) { tl.move (dx, dy); br.move (dx, dy); }
  void stretch (int dx, int dy) { br.move (dx, dy); }
```

```
};
/* Rotate shape 90 degrees. */
void rotate (void *shape) {
  switch ((shape_tag *) shape) {
    case s_point:
    case s_circle:
      break;
    case s_rectangle:
      {
        rectangle *rect = (rectangle *) shape;
        int d = ((rect->bot_right ().x_coord ()
                  - rect->top_left ().x_coord ()) -
                 (rect->top_left ().y_coord ()
                  - rect->bot_right ().y_coord ()));
        rect->move (d, d);
        rect->stretch (-2.0 * d, -2.0 * d);
      }
    }
}
```

a) Rewrite this so that each class has a Rotate method, and no tag field (i.e., write an object-oriented solution)
b) What if we add a Triangle class? What modifications would be necessary with the original version, and our new version?
c) Discuss the differences between changing the definition of the rotate method in the original and new (OO) version. (Remember that we have added the Triangle.)
d)

Problem 4

Consider the classes C and SC from the lecture slides.

We know that this language allows overloaded methods to be inherited, that is the scope for overloaded methods for a subclass includes the inherited methods.

Here is the answer to the question posed at the lecture (to which method are the different calls bound):

```
C c     = new C();
SC sc   = new SC();
C c'    = new SC();

c.equals(c)     //1     equals 1
c.equals(c')    //2     equals 1
c.equals(sc)    //3     equals 1

c'.equals(c)    //4     equals 1
c'.equals(c')   //5     equals 1
c'.equals(sc)   //6     equals 1
```

```
sc.equals(c)    //7     equals 1
sc.equals(c')   //8     equals 1
sc.equals(sc)   //9     equals 2
```

It is only in //9 that the equals 2 method is called, the reason being that overloading is resolved at compile time. The three calls to c' (even though the value of c' is a SC-object) will be calls to equals 1. //7 is also a call to equals 1, as the parameter c is of type C – same with //8.

The method equals 1 comes in two versions: the C_equals 1 and the redefined SC_equals 1.

a) Indicate for the above first 8 cases which of the equals 1 are called.

b) Now, suppose that class SC does not have the first equals method, the one with parameter of type C overriding the equals from class C. Determine which of the remaining methods is executed for each of these 8 cases:

```
c.equals(c)     //1
c.equals(c')    //2
c.equals(sc)    //3

c'.equals(c)    //4
c'.equals(c')   //5
c'.equals(sc)   //6

sc.equals(c)    //7
sc.equals(c')   //8
```

Problem 5

a) Write in Java both an abstract data type and a class for the data type Date, with year, month and day, operations before and after and daysBetween. In the abstract data type the operations before, after and daysBetween shall take two Dates, while the operations for the class Date shall have just one Date parameter.

b) There is on 'obvious' way of doing this, where Date is depending on how year, month and day is represented (e.g. as int variables). How would you make Date independent of this representation?