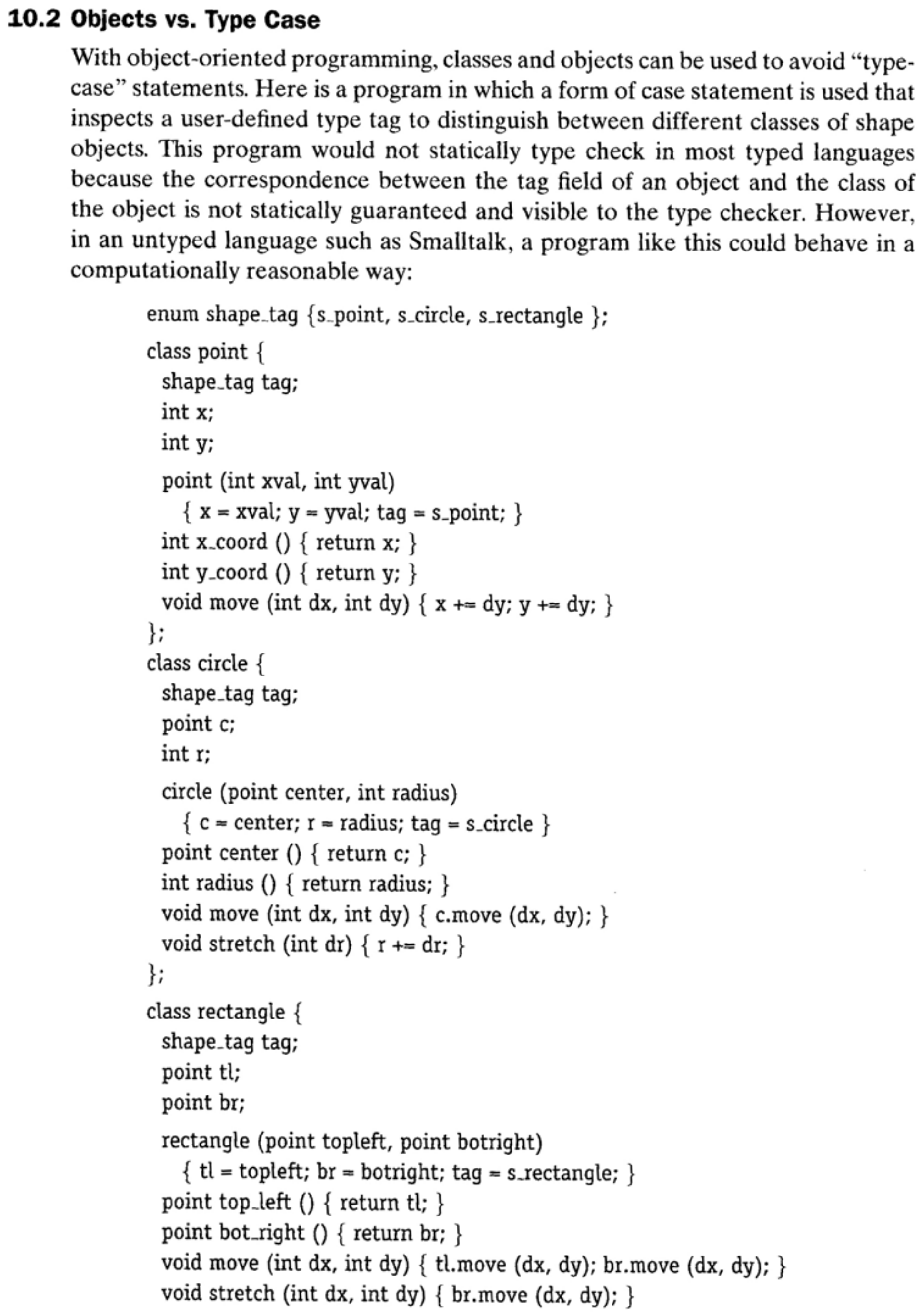
Weekly exercises in INF3110 – OO-II

2.11 – 6.11, 2009

Problem 1

Exercise 10.2 in Mitchell. Just sketch a solution.



10.2 Objects vs. Type Case

With object-oriented programming, classes and objects can be used to avoid "type-case" statements. Here is a program in which a form of case statement is used that inspects a user-defined type tag to distinguish between different classes of shape objects. This program would not statically type check in most typed languages because the correspondence between the tag field of an object and the class of the object is not statically guaranteed and visible to the type checker. However, in an untyped language such as Smalltalk, a program like this could behave in a computationally reasonable way:

```
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, instead of rotate being a function, each class has a rotate method and the classes do not have a tag.

(b) Discuss, from the point of view of someone maintaining and modifying code, the differences between adding a triangle class to the first version (as previously written) and adding a triangle class to the second [produced in part (a) of this question].

(c) Discuss the differences between changing the definition of rotate (say, from 90° to the left to 90° to the right) in the first and the second versions. Assume you have added a triangle class so that there is more than one class with a nontrivial rotate method.

Problem 2

Consider these two classes in a language that resembles Java:

```
class Rect {
  Point ul;       // upper left corner
  Point lr;       // lower right corner

  void setUL(Point newUL){ this.ul = newUL;};
};

class ColorRect extends Rect{
  ColorPoint ul;
  ColorPoint lr;
}
```

a) The intention with the redefinitions of the inherited ul and lr in class ColorRect is that the types of ul and lr shall be ColorPoint instead of Point. This is not allowed in Java. Any idea why?

b) How would this be done had Java had virtual classes?

A virtual class is a class (in a superclass) that can be redefined in the same way as methods can be redefined in subclasses, with the constraint that it can only be redefined to subclasses of the constraint the original class. In the example with virtual classes from the lecture (Point/Colopoint) this implies that the virtual class ThisClass can only be redefined to subclasses of Point, which is the constraint of ThisClass.

What would be the implication for type checking?

c) Virtual classes are not part of Java. Would casting help, like in:

```
class ColorRect extends Rect {
  void setUL(Point newUL){this.ul = (ColorPoint)newUL;};
}
```

Problem 3

a) Is there any alternative to multiple inheritance if only re-use of implementation is wanted?
Use the Stack, Queue and Dequeue example and try both to base Queue and Stack on Dequeue, and to base Dequeue on Stack and Queue.

b) What if one would also like to have the subtyping relationship, so that references typed with e.g. both Stack and Queue can denote objects of the resulting class?

Problem 4

A Java array of type T is declared by T[]. The Java subtype rule for array types is

S'[] subtype of S[] if S' subtype of S

Suppose we have class C with subclass CSub and that the method 'methodOfCSubOnly()' is defined in CSub only and not in C.

Look at

```
class TypeTest {
  C v = new C();
  void arrayProb(C[] anArray) {
    if (anArray.length >0)
      anArray[0] = v;                     // (2)
  };
  static void main(string[] args) {
    TypeTest tt = new TypeTest();
    CSub paramArray = new CSub[10];
    tt.arrayProb(paramArray);             // (1)
    paramArray[0].methodOfCSubOnly();     // (3)
  };
}
```

What happens? Especially at lines (1) (2) and (3).

Problem 5
Suppose that we have class Reservation with subclasses FlightReservation and TrainReservation as described in the foil set. As part of a reservation system it is desirable to have a collection of reservations that cater for possibly new subclasses for new kinds of reservations (e.g. for space travels).

How would you make a print method that prints all elements of such a collection, using the new generic mechanisms of Java?

Problem 6

We have seen structural type compatibility and subtyping wrt e.g. Smalltalk, as described in Mitchell. Objects of classes that have the same interface in terms operations if they provide the same set (or subset) of operations.

Consider the following Java sketch:

```
interface cowboy {void draw(); ...}
interface shape {void draw(); ...}

class LuckyLuke implements cowboy, shape {...}
```

Is this an example of structural (sub)typing, given the fact that Java may very well get the same method from different interfaces, but still only provide one implementation?

Problem 7

The FlightReservation class we have seen a couple of times has a Flight attribute. We assume that this is a reference to an object of class Flight. The Flight object represents the actual flight reserved.

In the flight table of SAS we have entries for e.g. SK451 (Oslo to Copenhagen). Suppose that we would like to represent such an entry by means of a FlightType object. Class FlightType would therefore have attributes that are common to all SK451 Flights, like source, destination, scheduled departure time (8.20), scheduled flying time (1.10), scheduled arrival time, etc.

SK451 takes place every day (or almost), so a reservation system would need to have one Flight object for each actual flight. These Flight objects will have a representation of seats (free, occupied), and for other reasons one may imagine that they will also have actual departure time, actual flight time and delay (departure and arrival delay).

It is perfectly possible to do this without inner classes, but if you should exploit inner classes, how would this be done. Of special interest is of course the functions computing the departure and arrival delays.

Feel free to be inspired by the slide on inner classes exemplified by class Apartment, specially the fact the attribute hight of Apartment is visible in the inner classes. Attributes like scheduled departureTime and arrivalTime should be attributes of the outer class FlightType.

Problem 8

Consider the classes C and SC at slide 29 from 19.10.2009.

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
```