Item 20: Replace unions with class hierarchies

The C `union` construct is most frequently used to define structures capable of holding more than one type of data. Such a structure typically contains at least two fields: a union and a *tag*. The tag is just an ordinary field used to indicate which of the possible types is held by the union. The tag is generally of some `enum` type. A structure containing a union and a tag is sometimes called a *discriminated union*.

In the C example below, the `shape_t` type is a discriminated union that can be used to represent either a rectangle or a circle. The `area` function takes a pointer to a `shape_t` structure and returns its area, or `-1.0`, if the structure is invalid:

```
/* Discriminated union */
#include "math.h"
typedef enum {RECTANGLE, CIRCLE} shapeType_t;

typedef struct {
    double length;
    double width;
} rectangleDimensions_t;

typedef struct {
    double radius;
} circleDimensions_t;

typedef struct {
    shapeType_t tag;
    union {
        rectangleDimensions_t rectangle;
        circleDimensions_t    circle;
    } dimensions;
} shape_t;

double area(shape_t *shape) {
    switch(shape->tag) {
      case RECTANGLE: {
        double length = shape->dimensions.rectangle.length;
        double width  = shape->dimensions.rectangle.width;
        return length * width;
      }
      case CIRCLE: {
        double r = shape->dimensions.circle.radius;
        return M_PI * (r*r);
      }
      default: return -1.0; /* Invalid tag */
    }
}
```


The designers of the Java programming language chose to omit the `union` construct because there is a much better mechanism for defining a single data type capable of representing objects of various types: subtyping. A discriminated union is really just a pallid imitation of a class hierarchy.

To transform a discriminated union into a class hierarchy, define an abstract class containing an abstract method for each operation whose behavior depends on the value of the tag. In the earlier example, there is only one such operation, `area`. This abstract class is the root of the class hierarchy. If there are any operations whose behavior does not depend on the value of the tag, turn these operations into concrete methods in the root class. Similarly, if there are any data fields in the discriminated union besides the tag and the union, these fields represent data common to all types and should be added to the root class. There are no such type-independent operations or data fields in the example.

Next, define a concrete subclass of the root class for each type that can be represented by the discriminated union. In the earlier example, the types are circle and rectangle. Include in each subclass the data fields particular to its type. In the example, radius is particular to circle, and length and width are particular to rectangle. Also include in each subclass the appropriate implementation of each abstract method in the root class. Here is the class hierarchy corresponding to the discriminated union example:

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    final double radius;

    Circle(double radius) { this.radius = radius; }

    double area() { return Math.PI * radius*radius; }
}

class Rectangle extends Shape {
    final double length;
    final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width  = width;
    }
    double area() { return length * width; }
}
```