Chapter 7

Data abstraction

Recall that a type is a collection of data values and the related operations. Programmer can define new data values defining structured data and new operations by defining functions on them. This model lacks following features.

coherence: One general principle of PL design is *coherence*: keeping related information together. We would like to keep both data and procedure together. An informal coherence technique is to keep the data and procedures in a single file. C has support for this. A more explicit language support for coherence is needed.

abstraction: Procedures are abstractions of computations. How can we abstract coherent units of data and functions? For example, C abstracts these in `.h` files.

visibility restriction: A `.h` file in C might contain the complete information. This might work for small programs written by a single programmer. This model is not sufficient for large scale software development by teams of programmers. In large software projects, complex relations (developer, tester, etc.) exist between team members and the full exposure of information may neither be necessary nor desirable.

Consider the example of a rectangle. A rectangle is characterized by its height and width (and center, if we want to draw it). In a language without structured data, we might say the following:

```
int height, width;
```

This ignores the underlying structure relating these variables. We would like to group data items that are related. This is the idea behind structured data types. In C, for example, we can group the data items.

```
struct rectangle {
   int height, width;
}
```

We can define some operations on this data type, like calculating the area.

```
float getArea(struct rectangle) {
         /*   calculations */
}
```

We would like to keep data and function definitions "together" where together means

1. A language construct which supports such grouping
2. Language support for separate and independent compilation

In Ada, for example, the language construct is called `package`. A package for rectangle looks as shown below.

```
package rectangle is   // package definition
   type rect is
      record
         width, height: float;
      end record;
   function rectangleArea(r: in rect) return float;
end rectangle;

package body rectangle is   // package body
   function rectangleArea(r: in rect) return float is
   begin
     ....
   end rectangleArea;
end rectangle;
```

Note that this keeps the data and procedure in a single unit. The users of the package know only of the package definition and nothing of the body. The body is known only to the implementor. Such a programming practice is called modular programming.
As mentioned before, it is a good practice to control the visibility of data values, in this case, the `width` and `height` of `rectangle`. Otherwise a careless user or malicious programmer can make the width of the rectangle negative.

```
    myRectangle: rectangle;
    myRectangle.width = -4;
```

This problem is acute especially in large programming projects involving tens to hundreds of programmers. Whose responsibility is it to keep the data in a consistent state: author's or user's? It is the author's responsibility. The author hides all the data. Otherwise, all routines operating on the rectangle need to check for such exceptional conditions. This overhead is due to the unrestricted access to data. Errors can be minimized it data operations are done through well-defined and controlled interfaces. Instead of allowing unrestricted access to data, we can restrict access to data only through well-defined and well-behaved interfaces.

In other words, we make data invisible to the user of the rectangle and allow only certain operations (like setting the dimensions, drawing, etc.) to the user. This is the principle of *data hiding* or *encapsulation*.
Program development proceeds from specification to implementation. The specification specifies the interfaces and constraints ("width cannot be negative"). The implementor is free to do anything as long as the specifications are met. To avoid the possibility of users using some implementation detail and to ensure that constraints are met, the author hides implementation details. The PL should provide mechanisms for this.
We can extend the above defintion of rectangle by making the data `private` as shown below.

```
package rectangle is
   type rect is private;
   procedure setRect(w, h: in float; r: in out rect);
   function rectangleArea(r: in rect) return float;
private
   type rect is
      record
         width, height: float;
      end record;
end rectangle;

package body rectangle is
   procedure setRect(w, h: in float; r: in out rect) is
   begin
     ... // check for error conditions
   end
   function rectangleArea(r: in rect) return float is
   begin
     ....  // no need to check for exceptions
   end rectangleArea;
end rectangle;
```

Note that the package specification part is the interface or contract and the package body is the implementation. Since the data is hidden, the package should provide mechanisms for *constructing* and *destroying* data values. We can think of `setRect` as a constructor, but more complicated constructors are needed in practice.
ML has a full complement of data hiding mechanisms. The interface part is called a *signature* and the implementation part the *structure*. A structure mainly consists of

1. types

2. constants

3. function definitions

4. exceptions

The structure for rectangle in ML looks as below

```
structure Rectangle = struct
    fun newRect(w, h:real) = (w, h);
    fun area(w, h:real) = w*h;
end;
```

Note that structures are implementations.
ML restricts visibility through *signatures.* Suppose we want to make rectangles read-only. Instead of changing the defintion of the structure, we define a different signature.

```
signature roRECT = sig    // signature defintion
   val area: (real*real) -> real;
end;

structure roRect:roRECT = Rectangle; // visibility restriction for Rectangle
```

Note that the inner structure of rectangle is visible to the user.

```
- val r=Rectangle.newRect(3.0, 4.0);
 ....
- r
val it = (3.0, 4.0): real*real
```

If the values are really hidden, we should not be able to access the values of `r`. ML has the type `abstype` which hides the values.

```
abstype rectangle = Rect of real*real // abstract data type
   with
        fun init(x:real, y:real) = Rect (x, y);
        fun area(Rect(x,y)) = x*y;
   end;

structure rectStr = struct  // not abstract
    datatype rectangle = Rect of real*real;

    fun init(x:real, y:real) = Rect (x, y);
    fun area(Rect(x,y)) = x*y;
end;

val a = init(3.0, 4.0);
val b = rectStr.init(3.0, 4.0);
area(a);
```

```
rectStr.area(b);
area(b); // error
rectStr.area(a); // error

- a;
val a = -: rectangle   // data value hidden
- b;
val b = Rect (3.0, 4.0): rectStr.rectangle   // data value not hidden
```

Note that the same structure can be viewed through two different signatures with different results.
The following example illustrates the use of *opaque signatures*.

```
signature SIG1 = sig   // interface
   val i: int;
   type t;
   val x: t;
   val y: t;
   val f: t*t -> bool;
end;

structure Struct1 = struct // implementation
   val i = 3;
   type t = int;
   val x = 4;
   val y = 5;
   fun f(a,b) = (a=b);
end;

structure Struct2: SIG1 = Struct1;  // Struct1 viewed through SIG1
structure Struct3:> SIG1 = Struct1;  // Struct1 viewed through "opaque" SIG1

open Struct2;
x;  // returns 4
f(x, y); // returns false
f(4, x); // returns true
open Struct3;
x;  // returns -
f(x, y); // returns false
f(4, x); // returns error, no probling allowed
```

Objects also provide a mechanism for data hiding. We can define a class `Rectangle` in Java as follows.

```
class Rectangle {
   private int height, width;
   private Point center;  // class "Point" is a field of Rectangle

   Rectangle() {   // default constructor
      height = 10;
      width = 10;
      center = new Point(0, 0);
   }

   Rectangle(int height, int width) throws IllegalValueException {

      if (height < 0) throw new IllegalValueException();
      if (width < 0) throw new IllegalValueException();

      this.height = height;
      this.width = width;
      center = new Point(0, 0);
   }

   public int getWidth() {
      return this.width;
   }

   public int setWidth(int w) throws IllegalValueException {
      if (w < 0) throw new IllegalValueException();
      width = w;
   }

   public void draw() {
       System.out.println("Drawing a rectangle");
   }
}
```

Note that access to data values is mediated through methods which can check for a variety of conditions.

In this chapter, we discussed some techniques for data abstraction: packages in Ada, signatures in ML, and classes in Java. One problem with the techniques discussed so far is that they are rigid. Suppose we want to add additional functionality to rectangle, like color. We need to define a new data type. The new datatype may *contain* the rectangle package or structure, but the fact that a colored rectangle is otherwise a rectangle is lost to the language. Object oriented programming techniques make this explicit.