Chapter 9
Data Abstraction

The physical machine manipulates data of only one type: bit strings. The types of a high-level language impose an organisation on this undivided universe, giving each value a sort of “wrapping”. Each value is wrapped in an encapsulation (its type) which provides the operations that manipulate it. The type system for a language establishes how transparent this encapsulation is. In type-safe languages, the encapsulation is completely opaque, in the sense that it does not allow access to the representation (or better: every access can only take place using or by means of the encapsulation itself).

The last chapter presented in detail many of the predefined types and the principal mechanisms for defining new ones. The latter, however, are fairly limited: finite homogeneous aggregations (arrays) and heterogeneous ones (records), recursive types and pointers. The operations possible on these composite types are predefined by the language and the programmer is restricted to use them. It is clear that the programmer, using just the mechanisms we discussed in Chap. 8, cannot really define a *new* type, when it is understood, using our definition, as a collection of (homogeneous and effectively presented) values equipped with a set of operations. The user of a language can only make use of the existing capsules and only has highly limited ways to define new types: there are few *data abstraction* mechanisms.

In this chapter, we will present some of the main ways in which a language can provide more sophisticated mechanisms for defining abstractions over data. Among these, we will discuss the so-called abstract data type, which in different forms is provided by several languages. We then look at modules, a largely similar concept, but which mainly applies to programming in the large. These abstraction mechanisms also constitute an introduction to some themes that we will encounter again with object-oriented programming. The key concepts that we will investigate in this chapter are the separation between *interface* and *implementation* and the concepts associated with *information hiding*.

9.1 Abstract Data Types

The introduction of a new type using the mechanisms discussed in the last chapter

```
type Int_Stack = struct{
                    int P[100];  // the stack proper
                    int top;     // first readable element
}
Int_Stack create_stack(){
   Int_Stack s = new Int_Stack();
   s.top = 0;
   return s;
}
Int_Stack push(Int_Stack s, int k){
   if (s.top == 100) error;
   s.P[s.top] = k;
   s.top = s.top + 1;
   return s;
}
int top(Int_Stack s){
   return s.P[s.top];
}
Int_Stack pop(Int_Stack s){
   if (s.top == 0) error;
   s.top = s.top - 1;
   return s;
}
bool empty(Int_Stack s){
   return (s.top == 0);
}
```

Fig. 9.1 Stack of integers

does not permit the user of a language to define types at the same level of abstraction as that enjoyed by the predefined types in these languages.

By way of an example, Fig. 9.1 shows one possible definition in our pseudo-language of the stack of integers data type. We assume a reference model for variables. When defining a type of this kind, it is probably intended that a stack of integers will be a data structure that can be manipulated by the operations of creation, insertion, access to the top element, removal of the top element. However, the language does not guarantee that these are the *only* ways in which a stack can be manipulated. Even if we adopt a strict type equivalence by name discipline (so that a stack is introduced by a declaration of the type `Int_Stack`) nothing prevents us from directly accessing its representation as an array:

```
int second_from_top()(Int_Stack c){
   return c.P[s.top - 1];
   }
```

From a general viewpoint, therefore, while languages provide predefined data abstractions (types) which hide implementations, the programmer cannot do this for themselves. To avoid this problem, some programming languages allow the definition of data abstractions which behave like predefined types as far as the (in)accessibility of representations is concerned. This mechanism is called an *abstract data type* (ADT). It is characterised by the following primary characteristics:

1. A name for the type.
2. An implementation (or representation) for the type.
3. A set of names denoting operations for manipulating the values of the type, together with their types.
4. For every operation, an implementation that uses the representation provided in point 2.
5. A security capsule which separates the name of the type and those of the operations from their implementations.

One possible notation for the stack of integers ADT in our pseudo-language could be that depicted in Fig. 9.2.

```
abstype Int_Stack{
   type Int_Stack = struct{
                        int P[100];
                        int n;
                        int top;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         Int_Stack s = new Int_Stack();
         s.n = 0;
         s.top = 0;
         return s;
      }
      Int_Stack push(Int_Stack s, int k){
         if (s.n == 100) error;
         s.n = s.n + 1;
         s.P[s.top] = k;
         s.top = s.top + 1;
         return s;
      }
      int top(Int_Stack s){
         return s.P[s.top];
      }
      Int_Stack pop(Int_Stack s){
         if (s.n == 0) error;
         s.n = s.n - 1;
         s.top = s.top - 1;
         return s;
      }
      bool empty(Int_Stack s){
         return (s.n == 0);
      }
}
```

Fig. 9.2 ADT for stacks of integers

A definition of this kind must be interpreted as follows. The first line introduces the name of the abstract data type. Line 2 provides the representation (or *concrete type*) for the abstract type `Int_Stack`. Lines 7 to 12 (introduced by `signature`) define the names and types of the operations that can manipulate an `Int_Stack`. The remaining lines (introduced by `operations`) provide the implementation of the operations. The important point of this definition is that inside the declaration of type, `Int_Stack` is a synonym for its concrete representation (and therefore the operations manipulate a stack as a record containing an array and two integer fields), while outside (and therefore in the rest of the program), there is no relation between an `Int_Stack` and its concrete type. The only possible ways to manipulate an `Int_Stack` are provided by its operations. The function `second_from_top()`, which we defined above, is now impossible because type checking does not permit the application of a field selector to an `Int_Stack` (which is *not* a record outside of its definition.)

An ADT is an opaque capsule. On the outside surface, visible to anyone, we find the name of the new type and the names and types of the operations. Inside, invisible to the outside world, there are the implementations of the type and its operations. Access to the inside is always controlled by the capsule which guarantees the consistency of the information it encloses. This external surface of the capsule is called the *signature* or *interface* of the ADT. On its inside is its *implementation*.

Abstract data types (in languages that support them, for example ML, and CLU[1]) behave just like predefined types. It is possible to declare variables of an abstract type and to use one abstract type in the definition of another, as has been done, by way of example, in Fig. 9.3. The code in Fig. 9.3 implements (very inefficiently) a variable of type integer using a stack. Inside the implementation of `Int_Var`, the implementation of `Int_Stack` is invisible because it was completely encapsulated when it was defined.

9.2 Information Hiding

The division between interface and implementation is of great importance for software development methods because it allows the separation a component's use from its definition. We also saw this distinction when we looked at control abstraction. A function abstracts (that is, hides) the code constituting its body (the implementation), while it reveals its interface, which is composed of its name and of the number and types of its parameters (that is, its signature). Data abstraction generalises this somewhat primitive form of abstraction. Not only is how an operation is implemented hidden but so is the way in which the data is represented, so that the language (with its type system) can guarantee that the abstraction cannot be violated. This phenomenon is called *information hiding*. One of the more interesting

[1]In object-oriented languages, we will see that it is possible to obtain the same abstraction goal using a similar but more flexible method.

Constructors, transformers and observers

In the definition of an abstract data type, `T`, the operations are conceptually divided into three separate categories:

- *Constructors*. These are operations which construct a new value of type `T`, possibly using values of other known types.
- *Transformers* or *operators*. These are operations that compute values of type `T`, possibly using other values (of the same ADT or of other types). A fundamental property of a transformer `t` of type $\mathtt{S}_1 \times \cdots \times \mathtt{S}_k$ `->` `Y` is that, for every argument value, it must be the case that $\mathtt{t(s}_1\mathtt{,\ldots,s}_k\mathtt{)}$ is a value constructable using only constructors.
- *Observers*. These are operations that compute a value of a known type that is different from `T`, using one or more values of type `T`.

An ADT without constructors is completely useless. There is no way to construct a value. In general, an ADT must have at least one operation in each of the above categories. It is not always easy to show that an operation is really a transformer (that is, that each of its values is in reality a value that can be obtained from a sequence of constructors).

In the integer stack example, `create_stack` and `push` are constructors, `pop` is a transformer and `top` and `empty` are observers.

```
abstype Int_Var{
   type Int_Var = Int_Stack;
   signature
      Int_Var create_var();
      int deref(Int_Var v);
      Int_Var assign(Int_Var v, int n);
   operations
      Int_Var create_var(){
         return push(create_stack(),0);
      }
      int deref(Int_Var v){
         return top(v);
      }
      Int_Var assign(Int_Var v, int n){
         return push(pop(v),n);
      }
}
```

Fig. 9.3 An ADT for an integer variable, implemented with a stack

consequences of information hiding is that, under certain conditions, it is possible to substitute the implementation of one ADT for that of another while keeping the interface the same. In the stack example, we could opt to represent the concrete type as a linked list (ignoring deallocation) allocated in the heap. This is de-

```
abstype Int_Stack{
   type Int_Stack = struct{
                       int info;
                       Int_stack next;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         return null;
      }
      Int_Stack push(Int_Stack s, int k){
         Int_Stack tmp = new Int_Stack(); // new element
         tmp.info = k;
         tmp.next = s;                    // chain on
         return tmp;
      }
      int top(Int_Stack s){
         return s.info;
      }
      Int_Stack pop(Int_Stack s){
         return s.next;
      }
      bool empty(Int_Stack s){
         return (s == null);
      }
}
```

Fig. 9.4 Another definition of the `Int_Stack` ADT

fined as in Fig. 9.4. Under certain assumptions, by substituting the first definition of `Int_Stack` by that in Fig. 9.4, there should be no *observable* effect on programs that use the abstract data type. These assumptions centre on what the clients of the interface expect from the operations. Let us say that the description of the semantics of the operations of an ADT is a *specification*, expressed not in terms of concrete types but general abstract relations. One possible specification for our ADT `Int_Stack` could be:

- `create_stack` creates an empty stack.
- `push` inserts an element into the stack.
- `top` returns the element at the top of the stack without modifying the stack itself. The stack must not be empty.
- `pop` removes the top elements from the stack. The stack must not be empty.
- `empty` is true if and only if the stack is empty.

Every client that uses `Int_Stack` making use of this specification *only*, will see no difference at all in the two definitions. This property has come to be called the *principle of representation independence*.

The specification of an abstract data type can be given in many different ways, ranging from natural language (which we have done) to semi-formal schemata, to completely formalised languages that can be manipulated by theorem provers. A specification is a kind of contract between the ADT and its client. The ADT guarantees that the implementation of the operations (which are unknown to the client) *matches* the specification; that is, all the properties stated in the specification are satisfied by the implementation. When this happens, it is also said that the implementation is *correct* with respect to the specification.

9.2.1 Representation Independence

We can state the representation independence property as follows:

> Two correct implementations of a single specification of an ADT are observationally indistinguishable by the clients of these types.

If a type enjoys representation independence, it is possible to replace its implementation by an equivalent (e.g., more efficient) one without causing any (new) errors in clients.

It should be clear that a considerable (and not at all obvious) part of representation independence consists of the guarantee that both implementations are correct with respect to the same specification. This can be hard to show, particularly when the specification is informal. There is, however, a weak version of the representation independence property that concerns only correctness with respect to the signature. In a type-safe language with abstract data types, the replacement of one ADT by another with the same signature (but different implementation) does not cause type errors. Under the assumptions we have made (of type safety and ADT), this property is a theorem which can be proved by type inference. Languages like ML and CLU enjoy this form of representation independence.

9.3 Modules

Abstract data types are mechanisms for programming in the small. They were designed to encapsulate *one* type and its associated operations. It is much more common, however, for an abstraction to be composed of a number of inter-related types (or data structures) of which it is desired to give clients a limited (that is, abstract) view (not all the operations are revealed and there is concealment of the implementation). The linguistic mechanisms which implement this type of encapsulation are called *modules* or *packages*. They belong to that part of a programming language dealing with programming in the large, that is with the implementation of complex systems using composition and assembly of simpler components. The module mechanism allows the static partitioning of a program into separate parts, each of which is equipped with data (types, variables, etc.) as well as with operations (functions, code, etc.). A module groups together a number of declarations (of data and/or functions) as well as defining visibility rules for these declarations by means of which a form of encapsulation and information hiding is implemented.

Considering semantic principles, there is not much of a difference between modules and ADTs, mostly the ability to define more than one type at a time (according to the definitions we have given, an ADT is a particular case of a module). Pragmatically, on the other hand, the module mechanism affords greater flexibility, both in the ability to state how permeable its encapsulation is (indeed, it is possible to indicate on an individual basis which operations are visible or to choose the level of visibility), or in the possibility of defining generic (polymorphic) modules (recall Sect. 8.8). Finally, module constructs are often related to separate compilation mechanisms for modules themselves.[2]

Even given the enormous variety among existing languages, we can state the important linguistic characteristics of a module by discussing the example shown in Fig. 9.5. It is expressed, as usual, in an suitable pseudo-language. First of all, a module is divided, as is an ADT, into a public part (which is visible to all the module's clients) and a private part (which is invisible to the outside world). The private part of a module can contain declarations that do not appear at all in the public part (for example, the `bookkeep` function inside `Queue`). A module can mention some of its own data structures in the public part, so that anyone can use or modify them (the variable `c` in `Buffer`, for example). A client module can use the public part of another module by *importing* it (the `imports` clause). In our example, `Buffer` has an additional import clause in its private part.

We will not continue with this discussion, both because we would have to go into the details of the mechanisms in a specific language, and because we will return to many points when considering object-oriented programming. To conclude, let us merely observe that the module mechanism is often associated with some form of parametric polymorphism which requires link-time resolution. In our example, we could have defined a buffer of type `T`, rather than a buffer of integer, making the definition generic and then suitably instantiating it when it is used. Figure 9.6 shows the generic version of the buffer example. It can be seen how the buffer and the code are both generic. When the private part of `Buffer` imports `Queue`, it specifies that it must be instantiated to the same type (which is not specified) as `Buffer`.

9.4 Chapter Summary

This short chapter has presented a first introduction to data abstraction, which turns on the key concepts of *interface*, *implementation*, *encapsulation*, *data hiding*.

[2]Keep in mind that modules and separate compilation are independent aspects of a language. In Java, for example, it is not the module (package, in Java terminology) which is the unit of compilation, rather the class is.

```
module Buffer imports Counter{
   public
      type Buf;
      void insert(reference Buf f, int n);
      int get(Buf b);
      Count c; // how many times buffer has been used
   private imports Queue{
        type Buf = Queue;
      void insert(reference Buf b, int n){
         inqueue(b,n);
         inc(c);
      }
      int get(Buf b){
         return dequeue(b);
         inc(c);
      }
      init_counter(c);    // module initialisation part
}
module Counter{
   public
      type Count;
      void init_counter(reference Count c);
      int get(Count c);
      void inc(reference Count c);
   private
      type Count = int;
      void init_counter(reference Count c){
         c=0;
      }
      int get(Count c){
         return c;
      }
      void inc(reference Count c){
         c = c+1;
      }
}
module Queue{
   public
      type Queue;
      inqueue(reference Queue q, int n);
      int dequeue(reference Queue q);
      ...
   private
      void bookkeep(reference Queue q){
         ...
      }
      ...
}
```

Fig. 9.5 Modules

```
module Buffer<T> imports Counter{
   public
      type Buf;
      void insert(reference Buf f, <T> n);
      <T> get(Buf b);
      Count c; //  how many times buffer has been used
   private imports Queue<T>{
      type Buf = Queue;
      void insert(reference Buf b, <T> n){
         inqueue(b,n);
         inc(c);
         }
      <T> get(Buf b){
         return dequeue(b);
         inc(c);
         }
}
module Counter{
   public
      type Count;
      void init_counter(reference Count c);
      int get(Count c);
      void inc(reference Count c);
   private
      type Count = int;
      void init_counter(reference Count c){
         c=0;
         }
      int get(Count c){
         return c;
         }
      void inc(reference Count c){
         c = c+1;
         }
      init_counter(c);    // module initialisation
}
module Queue<S>{
   public
      type Queue;
      inqueue(reference Queue q, <S> n);
      <S> dequeue(reference Queue q);
      ...
   private
      void bookkeep(reference Queue q){
         ...
         }
      ...
}
```

Fig. 9.6 Generic modules

From a linguistic viewpoint, we have presented the following:

- *Abstract data type* mechanisms.
- Mechanisms to hide information and their consequences; that is, the *Principle of representation independence*.
- *Modules* which apply the concepts of encapsulation to programming in the large.

All of these concepts are treated in more depth in texts on software engineering. As far as this book is concerned, they are devices to help understanding object-oriented programming which is the subject of the next chapter.

9.5 Bibliographical Notes

The concept of module probably appears for the first time in the Simula language [1, 4], the first object-oriented language (see Chap. 13). The development of modules in programming languages is due to the work of Wirth [6], which includes the Modula and Oberon [7] projects.

The concept of information hiding made its first appearance in the literature in a classic paper by Parnas [5]. Abstract data types originate in the same context, as a mechanism guaranteeing abstraction that is different from modules. Among the languages that include ADTs, the most influential is certainly CLU [3] which is also the basis for the book [2].

9.6 Exercises

1. Consider the following definition of an ADT in our pseudo-language:

```
abstype LittleUse{
   type LittleUse = int;
   signature
      LittleUse prox(LittleUse x);
      int get(LittleUse x);
   operations
      LittleUse prox(LittleUse x){
         return x+1;
         }
      int get(LittleUse x){
         return x;
         }
}
```

Why this type is useless?

References

1. G. Birtwistle, O. Dahl, B. Myhrtag, and K. Nygaard. *Simula Begin*. Auerbach Press, Philadelphia, 1973.
2. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, 1986.
3. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, 1977.
4. K. Nygaard and O.-J. Dahl. The development of the SIMULA languages. In *HOPL-1: The First ACM SIGPLAN Conference on History of Programming Languages*, pages 245–272. ACM Press, New York, 1978.
5. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
6. N. Wirth. The module: A system structuring facility in high-level programming languages. In *Language Design and Programming Methodology*. LNCS, volume 79, pages 1–24. Springer, Berlin, 1979.
7. N. Wirth. From Modula to Oberon. *Softw. Pract. Exp.*, 18(7):661–670, 1988.