More on Encapsulation

- A module could be a single procedure or function as discussed in the last chapter.
- A typical module is, however, usually a group of several *components* declared for a *common* purpose.
- These components could be types, constants, variables, procedures, functions, and so on. A module is said to encapsulate its components.
- To achieve a *narrow* interface to other modules, a module typically makes only a few components *visible* outside. Such components are said to be exported by the module. There may be other components that remain hidden inside the module, being used only to assist the implementation of the exported components.
- A package, which is simply a named group of declared components, takes us one step further from procedure/function abstractions to modular abstractions.
- A package is useful in *information hiding* and can be used in programming more specialized kinds of modules: namely *abstract data types*, *object classes* and *generics*.
- A package may be viewed as an encapsulated set of bindings (types, constants, variables, procedures, *etc*). The *how* is of concern only to the implementor. Chances are that the user and the implementor are different persons.

- A package typically consists of two parts. *E.g.* Modula-2, Modula-3, Oberon, Ada, ML
  - The package declaration declares only the *exported* components.
  - The package body contains declarations of any *hidden* components.
  - The bodies of procedures and functions, *even those exported*, must be kept in the package body.

  E.g. In Ada

```
package trig is
  function sin(x:Float) return Float;
  function cos(x:Float) return Float;
end trig;

package body trig is
  pi:constant Float := 3.1416;
  function norm(x:Float) return Float is
    ...
  function sin(x:Float) return Float is
    ...
  function cos(x:Float) return Float is
    ...
end;
```

  - Elaborating the above package will bind "`trig`" to the following encapsulated set of exported bindings
    $\{$`sin` $\mapsto$ a fn abstraction for approximating sine,
    `cos` $\mapsto$ a fn abstraction for approximating cosine$\}$
  - The binding for "`norm`" is hidden.

- In the following, we examine generic abstractions, which is a step forward from procedure/function abstraction. We will also see how to define ADTs, objects and object classes using packages and generic packages in Ada in the next chapter.
- A generic abstraction is an abstraction over a *declaration*. A generic abstraction has a body that is a declaration, and a generic instantiation is a declaration that will *produce bindings* by elaborating the generic abstraction's body. *E.g.* generic packages in Ada
- Generic packages can be *parameterized*.

```
generic
  capacity:in Positive;
package queue_class is
  procedure append(newitem:in Character);
  procedure remove(olditem:out Character);
end queue_class ;

package body queue_class is
  items:array (1..capacity) of Character;
  size,front,rear:Integer range 0..capacity;
  procedure append(newitem:in Character) is ...
  procedure remove(olditem:out Character) is ...
begin
  ...
end queue_class;
```

  - The *formal parameter* of the generic package is "`capacity`," and there are corresponding *applied occurrences* of "`capacity`" in the generic package's body

- The followings are *different instantiations* of the example generic package.

```
package line_buffer is new queue_class(120);
package terminal_buffer is new queue_class(80);
```

- Generic packages can also have type parameters.

```
generic
  capacity:in Positive;
  type Item is Private;
package queue_class is
  procedure append(newitem:in Item);
  procedure remove(olditem:out Item);
end queue_class ;

package body queue_class is
  items:array (1..capacity) of Item;
  size,front,rear:Integer range 0..capacity;
  procedure append(newitem:in Item) is ...
  procedure remove(olditem:out Item) is ...
begin
  ...
end queue_class;

package line_buffer is
    new queue_class(120,Character);

type Transaction is record ... end record;
package audit_trail is
    new queue_class(100,Transaction);
```

- If an abstraction is parameterized with respect to a *value*, the abstraction can use the argument value, even if nothing is known about that value except its type.

- Type parameters are fundamentally different, however. A type parameter denotes an *unknown* argument type. But nothing useful can be done with the type parameter *unless* something is known about the argument type—in particular, what *operations* are applicable to values of the argument type.

```
generic
  type Item is Private;
  type Sequence is
    array (Integer range <>) of Item;
  with function precedes(x,y:Item) return Boolean;
package sorting is
  procedure sort(seq:in out Sequence);
  procedure merge(seq1,seq2:in Sequence;
                  seq:out Sequence);
end sorting;

package body sorting is
  procedure sort(seq:in out Sequence) is
  begin
    ...
    if precedes(seq(j),seq(i)) then ...
    ...
  end;
  procedure merge(...)  is ...
end sorting;
```

- As illustrated in the example, formal type parameters can be *mutually dependent*. It is possible for one formal parameter to have a type that is itself a (type) parameter. Or one type parameter may depend on another.

- Instantiation of a parameterized generic package must be *consistent* with the *type dependency* specified.

```
type FloatSequence is
  array (Integer range <>) of Float;
package ascending is
  new sorting(Float,FloatSequence,"<=");
package descending is
  new sorting(Float,FloatSequence,">=");
```

- It is an *anomaly* that an Ada procedure or function abstraction cannot be parameterized with respect to another procedure or function abstraction. Many *simpler* languages allow this.
- It is *normal* that procedure and function abstractions cannot have type parameters. No *major* language allows this.