Chapter 8 - Data Abstraction and the Object-Oriented Model

In Chapter 7, the idea of procedural abstraction was introduced. This concept permits the abstraction of higher-level execution units to lower-level ones. Two important properties of this abstraction are encapsulation and parameterization. **Encapsulation** refers to the isolation of the operational details of a procedure from the environment where it is used. In particular, the invoking unit does not need to know the algorithms, data structures, or any other details of the procedure. **Parameterization** refers to the ability to create a generalized abstraction that has the flexibility to perform a range of activities based on the values of parameters. For example, a sort procedure typically has parameters representing the array and its size. This provides the procedure with sufficient generality to sort any array of any size.

In this chapter, we apply this same abstraction principle to data types. We have already discussed how aggregate data types can be abstracted to simple data types through the use of arrays and records and how some languages permit the parameterization of attributes such as the domain of arrays. In this chapter, we extend our consideration to abstract data types, which include the data type and the operations defined on that data type. We introduce the idea of encapsulation for data types—that is, separating the detailed description from the usage of the type. We also apply an extension of the concept of parameterization of types to the abstract data types. Special abstract data type definitions, called monitors, permit the specification of concurrent processes. We describe the implementation of abstract data types in the programming language Ada is briefly introduced. The object-oriented model for computation with special attention given to inheritance and polymorphism. The implementation of these features in the Java programming language is examined.

8.1 Abstract data types

An abstract data type is defined as a collection of data structures and operations abstracted into a simple data type. As an example, consider an abstract data type (ADT) called bst, which represents a binary search tree. It consists of objects that belong to the category binary search tree along with a set of useful operations that could be performed on such a tree. Some possible operations with their associated parameters are as follows:

```plaintext
bst initialize_tree();
//returns a new empty tree

boolean empty(bst tree);
//returns true if tree is empty and false otherwise

item_type root(bst tree);
//returns the value of the item at the root of tree.
// An exception is raised if tree is empty.

bst left_subtree(bst tree);
//returns the binary search tree that is the
// left subtree of the root of tree. An
// exception is raised if tree is empty.

bst right_subtree(bst tree);
//returns the binary search tree that is the
// right subtree of the root of tree. An
// exception is raised if tree is empty.

void insert_in_tree(item_type item,bst tree);
```
//inserts item into tree at the correct position.

These definitions are given in a pseudolanguage which, though similar to Java, does not exactly follow the syntax of any actual programming language.

This abstract data type could then be used to define and manipulate binary search trees without knowledge of the data structure used to represent \texttt{bst} or the algorithms used to implement the operations. For example, a procedure could be written to count the number of nodes in a \texttt{bst} as follows:

```plaintext
int count_nodes(bst tree) {
    if (empty(tree))
        return 0;
    else
        return count_nodes(left_subtree(tree)) +
               count_nodes(right_subtree(tree)) + 1;
}
```

The important point here is that the objects of an abstract data type can be used without any knowledge of their implementation. All operations on elements of the ADT must be defined as part of the abstract data type. No other operations on the ADT are permitted.

There are two major reasons for the use of abstract data types. The first is the independence of the use of the abstract data type from its implementation, which permits modification of the implementation without affecting the execution units where the abstract data type is used. The second reason is the integrity that is maintained by restricting access to the operations provided in abstract data types. For example, if the binary search tree was implemented using pointers, it would not be possible for an execution unit using \texttt{bst} to destroy the binary search tree structure of a \texttt{bst} object, because the unit does not have direct access to the data involved.

In the following sections, we see how languages implement this important feature. In so doing, we continue to illustrate data abstractions through the use of a pseudolanguage.

### 8.2 Encapsulation

#### 8.2.1 Two Approaches to Encapsulation

There are two approaches to the definition of an abstract data type. The first is an extension of the type definition to include the definition of operations. Figure 8.1 illustrates the implementation of \texttt{bst} in our pseudolanguage using this approach. Here we see that the definition of type \texttt{bst} is divided into three parts. The \texttt{export} section indicates those operations that are to be visible to the unit that uses the \texttt{bst} data type. The \texttt{structure} section defines the data type itself and corresponds to the normal definition of a data type. The \texttt{local} section defines all nonexported objects--including data types, variables, and procedures---that are a part of the implementation, but that are completely hidden from units outside of this definition. The \texttt{bst} ADT defined in Figure 8.1 exports the six procedures defined earlier, defines the data as consisting of three components, and locally defines the bodies of the six procedures.

The second approach to defining an ADT is more general in that it can be used to define entities other than an ADT as well. It consists of a collection of object definitions divided into two classes: those that are to be visible to external units and those that are not visible. When this format is used to define an ADT, the ADT itself is included as one of the exported objects defined within the collection of visible definitions, as opposed to having the separate \texttt{data} section as it did in the first approach.
Figure 8.1 ADT definition by type

type bst {
    export: {
        initialize_tree,
        insert_in_tree,
        empty,
        root,
        left_subtree,
        right_subtree;
    }

    data: {item_type root_item
        bst left;
        bst right;
    }

    local: {
        bst initialize_tree() {
            item_type := null;
        }

        boolean empty() {
            return (item_type == null);
        }

        boolean left_subtree(bst tree) {
            if (empty())
                throw bst_error;
            else
                return left;
        }

        boolean right_subtree(bst tree) {
            if (empty())
                throw bst_error;
            else
                return right;
        }

        bst insert_in_tree(item_type item){
            if (empty()) {
                root_item = item;
                left.initialize_tree();
                right.initialize_tree();
            }
            else if (item < root_item)
                left.insert_in_tree(item);
            else
                right.insert_in_tree(item);
        }

        item_type root(bst tree) {
            if (empty())
                throw bst_error;
            else
                return root_item;
        }
    }
}

Figure 8.2 ADT definition by collection

collection binary_search_tree is
{
    export: {
        type bst;
        void initialize_tree(bst tree);
        boolean empty(bst tree);
        bst left_subtree(bst tree);
        bst right_subtree(bst tree);
        void insert_in_tree(item:in item_type, bst tree);
        item_type root(bst tree) return item_type;
    }

    local: {type bst
        {
            root_item : item_type;
            left : bst;
            right : bst;
        }
        void initialize_tree(bst tree) {
            tree = null;
        }

        boolean empty(bst tree) {
            return (tree == null);
        }

        bst left_subtree(bst tree) {
            if (empty(tree))
                throw bst_error;
            else
                return tree.left;
        }

        bst right_subtree(bst tree) {
            if (empty(tree))
                throw bst_error;
            else
                return tree.right;
        }

        void insert_in_tree(item_type item, bst tree) {
            if (empty(tree))
                tree = new bst(item, null, null);
            else if (item < tree.root_item)
                tree.left.insert_in_tree(item);
            else
                tree.right.insert_in_tree(item);
        }

        item_type root(bst tree bst) {
            if (empty(tree))
                throw bst_error;
            else
                return tree.root_item;
        }
    }
}
Figure 8.2 sketches how our example might look using this collection approach. Here the bst is defined as an exported type, but its definition is deferred to the local section. This indicates that bst is to be visible outside; that is, objects can be bound to this type, but its implementation is to be hidden. In this case, an outside reference to root_item is not permitted, because the data of bst is not visible beyond the local environment. This approach can be used to define multiple types within a single collection or, if no types are in the export section, simply to export a collection of procedures.

In either case, the definition has two major parts: information that is visible to an external program unit and information that is hidden. The visible information may be variables, constants, procedure call templates, type definitions, or type names. The hidden information may include the definition of types that are named in the visible section, bodies of procedures whose call templates are listed in the visible section, and complete definitions of local objects such as variables, types, and procedures.

Some languages permit a block of code to be included in the hidden section that is to be executed to initialize a variable whenever one is declared to be of the specified abstract data type. This code could, for example, replace the initialize_tree procedure in Figure 8.1 and, therefore, eliminate the necessity of calling that procedure before a variable of type bst can be used. Some languages also include a termination block of code that is executed whenever the defining unit of an ADT variable is exited.

Another feature some languages add to an ADT environment is the implementation of local variables as own variables. These variables, although known only locally to the ADT, retain their values from one activation of ADT procedures to the next.

Discuss: What are the advantages and disadvantages to the use of the type definition form compared to the collection form of defining ADTs?

8.2.2 Example of Encapsulation -- Ada Packages

The programming language Ada contains extensive data abstraction capabilities. The package is the basic data abstraction construct and follows the model of the collection construct from Section 8.1. The package includes almost all of the features described in the previous sections including public and private declarations, initialization statements, information hiding, and parameterization of types.

8.2.2.1 Package definition

The definition of a package in Ada is divided into two parts: the package specification and the package body. Each of these is, in turn, divided into two parts. The specification has a visible and a private part, whereas the body has a declarative and an executable part. The body may also contain exception handlers, but we do not consider that feature in our discussion.

The general form of the specification of an Ada package is

```ada
package <identifier> is
    <visible-declarations>
[private
    <private-declarations>]
end [<identifier>];
```

The identifier specifies the package name, and if an identifier is included with the end statement, it must match the identifier following package. The visible declarations declare those objects of the package that are visible outside the package. These may include variables, types, procedures, and functions. In the case of procedures and functions, only the header is given here. The body of these entities must be written in the package body. Declarations in the pri-
vate section of the visible part specify properties of visible objects that are not to be visible outside the package. This concept is further discussed later.

The body of a package is declared after its specification. It defines all entities that are completely local to the package as well as the bodies of all procedures and functions, both visible and local. Its form is

```
package body <identifier> is
    <declarative-part>
    [begin
        <sequence-of-statements>
    end <identifier>;
```

Again, both identifiers must match those found in the corresponding specification. The declarative part contains the bodies of all visible procedures and functions as well as the bodies of all local procedures and functions, that is, those used inside the package but not visible there. It also contains declarations for any local types and variables. Variables declared in the package body and used in subprograms defined in the package body retain their values from one invocation of a package subprogram to another, and hence they have a location binding that spans the activation of the execution unit where the package is declared. The sequence of statements in the executable part of a package body specifies an initialization procedure that is executed when the execution unit containing the package’s definition is first activated. This can be used to initialize local variables, for example.

Figure 8.3 shows an example of an Ada package. In this package, the function `random` and the integer variable `seed` are visible to the execution unit that uses this package. The three variables that are declared locally in the package body retain their values from one call of `random` to the next. The visible variable `seed` retains its value as well. The difference between a visible variable such as `seed` and the other local variables is that `seed` can also be accessed and changed by an external execution unit. The function `get_time_in_seconds` is local, because it is not included in the package specification but only in the package body. This means that it is not visible to any external execution unit.

Packages containing no body may be specified. Such packages simply provide a set of declarations to the external execution unit. Procedures and functions cannot be included in bodyless packages. An example of such a package is given in Figure 8.4. It defines the type `employee_record`, and the inclusion of this package would make this data type available to an external execution unit.

### 8.2.2.2 Private types

Frequently, an abstract data type is created inside a package that must be visible outside the package as a type, but the actual structure of the type must be hidden from the external unit. This prevents the user of the package from manipulating objects of this type directly, enforcing the use of the encapsulated operations. In Ada, this capability is provided through the use of **private types**.

A visible type whose definition is to be hidden is declared to be private in the visible part of the specification of the package. Its full definition is then given in the private part of the specification. Any additional declarations needed to fully determine the private type are also included in the private part, such as the type definitions for components of the private type.

Consider the example in Figure 8.5. A private type has its full definition specified within the body of the package where it is declared. Outside the package body, the structure of the type is unknown, although its name is visible. The only operations permitted outside the package on a private type are assignment, tests for equality and inequality, and visible operations defined by the package.

In the example from Figure 8.5, if a variable `tree` is of type `bst`, then `tree.root_item` would be legally accessible within the package body but not in an external execution unit, because the internal structure of type `bst` would not be visible.

A further restriction can be applied in Ada if a type is declared to be **limited private**. These types...
behave just like private types except even assignment and equality/inequality tests are disallowed outside the package. Only operations defined in the package body are permitted on limited private types.

**Figure 8.3 Ada package for random number generation**

```ada
package rng is
  function random return float;
  seed: integer;
end rng;

package body rng is
  modulus: integer := 65536;
  mult: integer := 13849;
  addon: integer := 56963;

  function random return float is
    begin
      seed := (seed*mult+addon) mod modulus;
      return float(seed)/float(modulus);
    end random;

  function get_time_in_seconds return integer is
    begin
      ...
    end get_time_in_seconds;

begin
  seed := get_time_in_seconds mod modulus;
end rng;
```

**Figure 8.4 Ada package without a body**

```ada
package employee is
  type employee_record is record
    name : string(1..20);
    id : integer;
    address : string(1..20);
    pay_rate : float;
  end record;
end employee;
```

**Figure 8.5 Ada package specification using private type**

```ada
package binary_search_tree is
  type bst is private;
private
  type bst is access node;
  type node is record
    root_item : item_type;
    left : bst;
    right : bst;
```
end record;
end binary_search_tree;

8.2.2.3 Using packages

A package in Ada either can be included in the unit containing the execution unit where it is used or can be
placed in a user library. If it is placed in a library in compiled form, it is made available to an execution unit by
appearing in a with statement of the form

    with <package-name>;

Visible names in a package must be preceded by a prefix indicating the package name when used within an
external unit. It is possible to remove this requirement through a use statement. This statement has the form

    use <package-name>;

and permits access to visible objects in the package without use of the prefix. Consider the package defined in Figure
8.5. If a program using this package declared the variable tree to be of type bst, the declaration would be

    tree:binary_search_tree.bst;

If a use clause were present, the prefix would be unnecessary. Therefore,

    use binary_search_tree;
    tree:bst;

is equivalent to the previous declaration of tree. Of course, if there are several visible objects with the same name
whose identity cannot be otherwise determined by context, a prefix might still be necessary to determine the originat-
ing package.

8.2.3 Encapsulation in C++

The programming language C++ uses the type model for data abstraction through the use of the struct con-
struct. The struct is seldom used in C++, because its capabilities are a subset of those of the object-oriented class
structure. We use the struct in this section, however, because we only need those features included in this subset to
illustrate encapsulation as described in Section 8.2.1.

The C++ struct definition includes a datatype and the functions that apply to that datatype. It also
may include the definition of any types and constants used in the datatype. All components of a C++ struct are
classified as private or public depending on whether they are visible only to struct functions or are visible every-
where. The general form of a C++ struct definition is

    struct <identifier>
    { public:
       <visible_definitions>
    private:
       <private_definitions>
    }

In order to observe how the C++ struct implements the type model of data abstraction, we consider the
binary search tree example implemented as a C++ struct in Figure 8.6. In this example, the data part of the
struct is defined as private and includes the definition of an auxiliary struct type named node. This node type includes an item_type entry containing the root value as well as pointers to the two subtrees of the root. We assume that item_type is defined elsewhere and can be either a type or a class in C++. The only data object belonging to the bst struct is a pointer to one of these node items. Because the node type and the pointer to a node component, named data, are both found in the private section of the bst definition, neither are visible to any execution units not contained within the struct definition.
Figure 8.6 BST abstract data type in C++

```c
struct bst {
private:
    typedef struct node {
        item_type root_item;
        bst* left;
        bst* right;
    } node;

    node* data;

public:
    bst() { data = NULL; }
    ~bst() { if (data!=NULL) { delete(data->left); delete(data->right); delete(data); } }
    int empty() { return (data==NULL); }
    bst left_subtree() { if (data!=NULL) return *(data->left); else throw; }
    bst right_subtree() { if (data!=NULL) return *(data->right); else throw; }
    void insert_in_tree(item_type item) {
        if (empty()) {
            data = new(node);
            data->root_item = item;
            data->left = new bst;
            data->right = new bst;
        } else if (item < data->root_item) (data->left)->insert_in_tree(item);
        else (data->right)->insert_in_tree(item);
    }
    item_type root() {
        if (data==NULL) throw;
        else return data->root_item;
    }
};
```

Figure 8.7 Example of Restrictive Monitor

```c
monitor restrict is

parameters:
    type item_type;
    lower,upper : integer;
end parameters;

export:
    type restricted_array is private;
    procedure addup(A:in restricted_array) result item_type;
    procedure store(VALUE:in item_type; LOCATION:in integer; A:in out restricted_array);
end export;

local:
    type restricted_array is
        array [lower..upper] of item_type;
    procedure addup(A:in restricted_array) result item_type is
        begin
            ...
        end addup;
    procedure store(VALUE:in item_type; LOCATION:in integer; A:in out restricted_array) is
        begin
            A[LOCATION] := VALUE;
        end store;
end local;
```

Figure 8.8 ADT Definitions with parameters

Type Definition from Figure 8.1
```c
type bst {
    parameters {
        type item_type;
    }
    export {
        ...
    }
};
```

Collection Definition from Figure 8.2
```c
collection binary_search_module {
    parameters {
        type item_type;
    }
    export {
        ...
    }
};
```
Because the C++ struct uses the type model, bst elements are defined by standard C++ variable declarations such as

```cpp
bst a;
```

which declares a to be a variable of type bst.

C++ also provides the ability to specify execution units that are activated upon the birth and the death of an instantiation of the data abstraction. These are called a constructor and a destructor, respectively.

The constructor is executed every time a new bst item is created by means of a declaration or by means of a new operation. Declaration occurs as shown above in the declaration of a. The new operation is used to dynamically create a bst item that is referenced by a pointer. For example, the sequence of statements

```cpp
bst* p;
p = new bst;
```

first declares p to be a pointer to a bst item with the second statement creating a new bst item and setting p so that it points to that new item. A constructor is an execution unit whose name is the name of the data type being defined. The constructor may have parameters, and any number of constructors can be defined as long as their parameter protocols are different. The body of a constructor specifies the code that is executed when an item is created. The constructor in Figure 8.6 specifies that a bst is initialized to have its data component contain the NULL pointer. Although no other constructor is specified here, one could be added that provides an item of item_type as a parameter and creates a one-element bst with that item at its root. This would be defined as follows:

```cpp
bst(item_type item) {
    data = new(node);
    this->insert_in_tree(item);
}
```

A bst with its root equal to x of item_type could then be created by

```cpp
bst a(x);
```

or

```cpp
p = new bst(x);
```

A C++ destructor is executed whenever a struct item ceases to exist, just prior to the demise of the item. This may occur when execution flow leaves the scope of the item as in

```cpp
{ ...
bst a;
...
} // destructor for bst a is executed upon leaving this block
```

A destructor is also executed when an item created by a new operation is disposed of through a delete operation. This is illustrated by

```cpp
bst* p;
...
p = new bst;
...
delete(p); // destructor for bst pointed to by p is executed here.
```
The name of the destructor is the struct name prefixed by a tilde (~). In the case of our bst struct, this name is ~bst. The bst destructor in Figure 8.6 calls the destructor on the two subtrees before calling the destructor on the root item of the tree.

8.2.4 Monitors--Concurrent Encapsulation

One strategy for concurrent processing is the use of a monitor. In this section, we examine the monitor as an encapsulation similar to that used in defining ADTs. The purpose of the monitor is to guarantee mutual exclusion in data access and/or processing among concurrently executing processes.

A monitor is similar to the collection that we defined earlier in this chapter in that it contains an encapsulated set of procedures and data objects. In its most restrictive form, a monitor might only permit one of its procedures to be in execution at a time. Because these procedures are called by concurrent processes and all manipulation of shared data is encapsulated inside the monitor, it is guaranteed that no two processes access the shared data concurrently. In the example in Figure 8.7, written in an Ada-like pseudolanguage, two exported procedures are provided, `addup` and `store`. The monitor differs from a collection as defined before in that these procedures are candidates for overlapping execution by concurrent processes. If this occurs, the first procedure invoked prevents any further invocations of procedures in the monitor from proceeding until the initial one is terminated. The rejected invocations are placed on a queue and are successively permitted to proceed one execution at a time when the currently executing invocation terminates. This structure guarantees that data of a type defined locally in the monitor can be accessed by only one process at a time, thus ensuring mutual exclusion.

This form of concurrency control may be overly restrictive for some applications. For this reason, a more flexible type of monitor is permitted by some languages that allows more than one process to execute procedures from the monitor concurrently, but provides some synchronization mechanism these procedures can use among themselves. This strategy results in all synchronization occurring within monitor procedures, so as to encapsulate synchronization as well as data and procedures. Procedures within this more flexible monitor are therefore able to execute concurrently.

Discuss: What would be some considerations in implementing a monitor?

Discuss: Why is encapsulation an important property of object-oriented languages?

Research: Devise a way to use encapsulation in a language that does explicitly support the idea. Rewrite Figure 8.3 using your method.

8.3 Parameterization

The parameterization of data types was introduced in Chapter 5, where the bounds on the domain of an array were considered as parameters. In this section, we extend this idea by permitting much broader parameterization of aggregate types.

8.3.1 Binary Search Tree with Parameterization

Our binary search tree ADT defined in Figures 8.1 and 8.2 presents a likely candidate for parameterization of `item_type`. This would permit our ADT to represent binary trees, where the nodes are of any defined type that is specified by an actual parameter.

To implement this, an additional section to specify parameters is required in the ADT definition. Figure 8.8 shows modifications of Figure 8.1 and 8.2 that could specify this feature.

Binding actual parameters to formal parameters is accomplished differently by the two different forms of expressing ADTs. The type definition form can simply use parameterized type names in variable declarations--for example,
bst(real) A;
bst(small_array) B;
bst(string) C;
bst(employee_record) D;

The collection form requires **instantiations** of the collection for each parameter set applied, which means the parameterized collection is just a shell. When the parameters are filled in, actual collections are created or instantiated. For example, constructs like the preceding ones could be expressed by

```plaintext
collection bst_real is bst(real);
collection bst_small_array is bst(small_array);
collection bst_string is bst(string);
collection bst_emp_rec is bst(employee_record);
```

Then the preceding variable declarations would be replaced by

```plaintext
bst_real.bst A;
bst_small_array.bst B;
bst_string.bst C;
bst_emp_rec.bst D;
```

In this case, because four actual collections are created containing the type bst, when one of the types is used, it must be prefixed by the appropriate collection name. This rule applies not only to types but to all visible objects defined in the instantiated collections.

Objects other than types could be defined parametrically in ADTs. For example, a procedure or an operation could be a parameter to the definition of an ADT. In the `binary_search_tree` definition, we have assumed the operation `<` makes sense for the parameterized type `item_type`. If, however, `item_type` is a record or an array, this operation has no built-in definition. Therefore, such types are invalid as actual parameters as our definition now stands despite the fact that in practice, nodes of a binary tree are frequently records.

One way to handle this problem is to change `<` to a function and have that function passed as a parameter when the collection is instantiated. Figure 8.9 illustrates the definition of collection `binary_search_tree` under this assumption, showing only the changes from Figure 8.2. Value returning procedure `LESS_THAN` is a formal parameter to the collection, a parameter that compares two values of `item_type`.

**Figure 8.9 ADT definition with operator parameter**

```plaintext
collection binary_search_tree {

    parameters {
        type item_type;
        boolean LESS_THAN(item_type A,item_type B);
    }

    export {
        ...
    }

    local {
        ...
    }

```
else if (LESS_THAN(item, tree.root_node))
    ...
else if (LESS_THAN(item, tree.root_node))
    ...
}
}

LESS_THAN takes on the usual definition for numeric or string types. For example, for type float the following function would be appropriate:

```c
boolean LESS_THAN_FLOAT(float X, float Y)
{
    return (X < Y);
}
```

For arrays or records, a more complex definition might be necessary. Suppose we wish to define two orders on employee_record type as follows:

```c
boolean LESS_NAME(employee_record X, employee_record Y)
{
    return X.NAME < Y.NAME;
}

boolean LESS_ID(employee_record X, employee_record Y)
{
    return X.ID < Y.ID;
}
```

Then we could instantiate two different collections with the statements

```c
collection bst_by_name is bst(employee_record, LESS_NAME);
collection bst_by_id is bst(employee_record, LESS_ID);
```

and declare the variables

```c
bst_by_name.bst X;
bst_by_id.bst Y;
```

Then X is a binary tree of employee records ordered by the name field of the record, and Y is a binary tree of employee records ordered by the id field.

If a language permits the overloading of procedures and procedure names are chosen appropriately, the use of procedures as parameters is avoidable if procedure names are chosen appropriately. This enforces a restriction on the execution unit using the ADT that we would rather avoid--namely, the requirement of using a specific name for the given procedure. This result requires external knowledge of the name used within the ADT definition. In the case of our preceding example, if all types that are actual parameters corresponding to item_type have a LESS_THAN procedure defined appropriately for them, LESS_THAN need not be passed as a parameter, because overloading assumes the appropriate LESS_THAN is chosen for each type.

A further extension of this idea is to permit operators such as < to be given overloaded definitions. Then artificial functions would not be necessary to redefine operators for types where the operator is already built into the language. Note that such overloading also precludes the use of two different orderings on the same type, as we did with employee_record earlier.
8.3.2 Example of Parameterization -- C++ Templates

Types in C++ can be parameterized through the use of the template feature. This is illustrated in Figure 8.10 by the use of the template header placed on the struct definition. Other than this header, the definition is identical to Figure 8.6. The template header specifies the name of the parameter (or parameters), \( \text{item	extunderscore type} \) in this case, and the type of the parameter. In Figure 8.10, the parameter is specified as \( \text{class} \), which means it can be any type or class in C++, and determines the type or class of the items that can be placed into the tree. The only restriction is that the type or class that is given as the actual parameter must have the binary operator \(<\) defined for it so the comparison can occur in the \( \text{insert	extunderscore in	extunderscore tree} \) function. Since C++ allows overloading of operators like \(<\), it is easy to construct types and classes that satisfy this requirement.

Templated types can then be instantiated in the following way:

\[
\begin{align*}
\text{bst<int> inttree;} \\
\text{bst<double> doubltree;} \\
\text{bst<String> Stringtree;} \\
\text{bst<Fraction> Fractiontree;}
\end{align*}
\]

where the first two \( \text{bst} \) declarations construct \( \text{bst}s \) on built-in C++ types and the last two construct \( \text{bst}s \) on user-defined classes. All of these \( \text{bsts} \) then accept calls on the provided functions such as

\[
\begin{align*}
\text{inttree.insert	extunderscore in	extunderscore tree}(6); \\
\text{doubltree.insert	extunderscore in	extunderscore tree}(12.5); \\
\text{String S(“abc”);} \\
\text{Stringtree.insert	extunderscore in	extunderscore tree(S);} \\
\text{Fraction F(12,5);} \\
\text{Fractiontree.insert	extunderscore in	extunderscore tree(F);} \\
\end{align*}
\]

\textbf{Figure 8.10 \( \text{bst} \) in C++ with template}

\begin{verbatim}
template <class item_type>
struct bst {
    private:
        typedef struct node {
            item_type root_item;
            bst* left;
            bst* right;
        } node;

        node* data;

    public:
        bst() { data = NULL; }
        ~bst() { if (data!=NULL) 
                  delete(data->left);
                  delete(data->right);
                  delete(data); }

        int empty() { 
            return (data==NULL);
        }

        bst left_subtree() { 
            if (data!=NULL)
                return *(data->left);
            else
                throw;
        }

        bst right_subtree() { 
            if (data!=NULL)
                return *(data->right);
            else
                throw;
        }

        void insert_in_tree(item_type item) { 
            if (empty()) { 
                data = new(node);
                data->root_item = item;
                data->left = new bst;
                data->right = new bst;
            } else if (item < data->root_item) 
                (data->left)->insert_in_tree(item);
            else 
                (data->right)->insert_in_tree(item);
        }

        item_type root() { 
            if (data==NULL)
                throw;
            else
                return data->root_item;
        }

    };
\end{verbatim}
8.3.3 Example of Parameterization -- Ada Generics

Ada permits the parameterization of packages through the use of a construct known as the **generic package**. The language also permits generic procedures and functions, but we do not discuss them here.

The general form of a generic package declaration is

```
generic
  <declaration-of-formal-parameters>
<package-specification>
```

The formal parameters for an Ada generic package may be data objects, types, or subprograms. These formal parameters may be used naturally within the package specification and body. The generic package does not, however, create an actual package but rather creates a template for a set of packages. When actual parameters are provided for the formal parameters, an instantiation occurs, creating an actual package. Many actual packages can be created from a single generic package. The construct for creating or instantiating a package from a generic is

```
package <package-name> is new <generic-name> [(<actual-parameter-list>)];
```

The best way to get a feel for the use of generics is to consider an example. Figure 8.11 is a generic package for the binary search tree. The two parameters for this generic are the type of the items in the tree and the definition of the operator function `<`. The latter is necessary because `item_type` does not necessarily have a built-in `<` operator. This use of generic permits, for example, the use of record types for `item_type` as long as an appropriate `<` is defined for the record and used as the actual parameter. The default definition of formal parameter `<` given by the `<>` box means that if the formal parameter `item_type` has an operator already defined by `<` and the second actual parameter is omitted, that operator can be employed by the package.

**Figure 8.11 Ada generic package definition**

```ada
generic
type item_type is private;
  with function "<" (A,B:item_type) return boolean is <;>
package binary_search_tree is
type bst is private;
  procedure initialize_tree(tree:out bst);
  function empty(tree:in bst) return boolean;
  function left_subtree(tree:in bst) return bst;
  function right_subtree(tree:in bst) return bst;
  procedure insert_in_tree(item:in item_type; tree:in out bst);
  function root(tree:in bst) return item_type;
private
type node is record
  root_item:item_type;
  left,right:bst;
end record;
type bst is access node;
end;

package body binary_search_tree is

bst_error:exception;

procedure initialize_tree(tree:out bst) is
begin
  tree:=NULL;
```
end initialize_tree;

function empty(tree:in bst) return boolean is
begin
  return tree=NULL;
end empty;

function left_subtree(tree:in bst) return bst is
begin
  if tree=NULL then
    raise bst_error;
  else
    return tree.left;
  end if;
end left_subtree;

function right_subtree(tree:in bst) return bst is
begin
  if tree=NULL then
    raise bst_error;
  else
    return tree.right;
  end if;
end right_subtree;

procedure insert_in_tree(item:in item_type; tree:in out bst) is
begin
  if tree=NULL then
    tree:=new node'(item,NULL,NULL);
  elsif item < tree.root_item then
    insert_in_tree(item,tree.left);
  else
    insert_in_tree(item,tree.right);
  end if;
end insert_in_tree;

function root(tree:in bst) return item_type is
begin
  return tree.root_item;
end root;

end binary_search_tree;

package bst_integer is new binary_search_tree(integer);
function less_id(A,B:in emp_rec) return boolean is
begin
    return A.id<B.id;
end less_id;

function less_name(A,B:in emp_rec) return boolean is
    pos:integer;
begin
    pos := A.name'first;
    loop
        exit when pos > A.name'last;
        exit when A.name(pos) /= B.name(pos);
        pos:=pos+1;
    end loop;
    if pos>A.name'last then
        return FALSE;
    else
        return A.name(pos)<B.name(pos);
    end if;
end less_name;

package bst_name is new binary_search_tree(emp_rec,less_name);
package bst_id is new binary_search_tree(emp_rec,less_id);

Figure 8.12 shows some sample instantiations of this generic package. The instantiation of bst_integer does not require a second parameter, because < is already defined for integer type. The two packages bst_name and bst_id require explicit functions for the second actual parameter. Notice that the two instantiations use the same item_type (employee_record) but different < functions, one ordering the tree on the component name and the other on the component id.

Trees might then be declared in an execution unit where all three packages are instantiated by

```
tree1 : bst_integer.bst;
tree2 : bst_name.bst;
tree3 : bst_id.bst;
```

Notice that prefixes are required on these type names, because all three instantiated packages define a type bst. In addition, the tree initialization procedures could then be invoked by

```
initialize_tree(tree1);
initialize_tree(tree2);
initialize_tree(tree3);
```

In this case, prefixes are not required, because the version of initialize_tree can be determined by the type of its actual parameter by overloading.

**Discuss:** While parameterized ADT definitions are a very interesting concept, their implementation can force other, perhaps undesirable, constraints on a compiler and run-time system. An implication of the implementation of generic packages in Ada is that binding of generic packages is dynamic—that is, packages must now be instantiated. This, in turn, makes the resolution of overloading a run-time process, not a static one.

a. Why does dynamic instantiation force dynamic overloading resolution?

b. Why are these implementation constraints unfortunate?
c. Can you dream up any way around these constraints? You may restrict the semantics of the language as you think about this one.
d. Can you think of any other implications of generic implementation?

Discuss: Consider the concepts of encapsulation and parameterization discussed at the beginning of this chapter. How well does the Ada package mechanism implement these concepts?

Reinforce: What would be possible parameters for each of the following ADTs?
a. stacks
b. polynomials
c. strings
d. sets

Reinforce: Consider the following Ada package specification:

```ada
generic
  size: integer;
  type set_type is private;
package SET_PACK is
  type set is private;
  function new_set return set;
  procedure add(s: in out set; item: in set_type);
  function member(s: in set; item: in set_type) return boolean;
  procedure delete(s: in out set; item: in set_type);
private
  type set is array (1..size) of set_type;
end SET_PACK;
```

The package implements a set ADT. Use this package to provide solutions to the following:
a. Declare two sets, intset and floatset, that are sets of 100 integers and 50 booleans respectively. Do this problem with and without the use statement. Does it make a difference?
b. Using your declarations, write code fragments to add 10 to the intset and 1.5 to the floatset. Do this problem with and without the use statement. Does it make a difference?
c. Add a new function, call it member, that checks to see if two distinct elements are in a set and takes three arguments. Is member a permissible name? (Hint: Yes it is. Why?) What does the argument list look like?
d. In the previous exercises, rewrite the code as if you were to use the operator “+” for “add”, the operator “-” for “delete”, and the operator “==” for “member”. Evaluate your code and give your assessment of whether it is better or worse than your original code.

Reinforce: Java does not allow the use of templates. Rewrite the Ada package SET_PACK from the previous exercise in Java. Use this class to provide the solutions to the following:
a. Declare two sets, intset and floatset, that are sets of 100 integers and 50 floats respectively. How did this differ from the Ada version?
b. Using your declarations, write code fragments to add 10 to the intset and 1.5 to the floatset. How does it differ from the Ada version?
c. Add a new function, call it member, that checks to see if two distinct elements are in a set (takes three arguments). How does this differ from the Ada version?

Reinforce: Consider the ADT specification below. STRINGS_PACK is an Ada generic package.

```ada
generic
  size: integer;
package STRINGS_PACK is
  type string is private;
  function length(s: in string) return integer;
private
  type string is array (1..size) of string character;
end STRINGS_PACK;
```
string_length: integer;

type string_range is private;

package STRINGS_PACK is

type string is private;

procedure assign(str1: out string; str2: in string);

procedure append(str: in out string; c: in character);

procedure concat(str1: in out string; str2: in string);

function index(str: in string; c: in character) return integer;

private

type string is array(1..string_length) of string_range;

end STRINGS_PACK;

a. Write a section of Ada code that declares a string of only capital letters of maximum size 100 characters long and initializes it to some value. Assume the package above is declared separately from and external to the code you write.

Hint: The statement

  type XYZ is range ('A'..'Z');

defines a type XYZ of capital letters.

b. Redo part (a), but use different statements/declarations so that you can use an alternate notation to make the references to the preceding package/modules.

Reinforce: Early versions of C++ compilers implemented templated classes by simply using text substitution of the template type name by the actual template type given in the declaration. Give some code fragments (Ada or C++ or Java) that show this is a good approach and that this is a bad approach.

Research: Devise a way to use parameterization in an imperative language.

8.4 Inheritance

In order to meet the data abstraction goal of reusability, programming languages have adopted the concept of inheritance. Inheritance permits one data type to implicitly inherit both the structure and the actions of another data type. Inheritance is associated with the object-oriented model of computing, because it was in this context that it came into widespread use. For this reason, we shift our terminology here from ADT to the object-oriented version of an ADT, the class.

8.4.1 The Concept of Inheritance

Within object-oriented languages, the concept of inheritance comes into play when some class A has a second class B derived from it. In this context, class B is said to inherit from class A. What class B inherits is A’s internal state structure and the actions that A specifies in terms of message/method pairs. We will adopt the terminology that class A is called the parent class and class B, which is derived from A, is called the derived class.

In general, there are three possible ways the derived class can modify its parent class.
1. Add new components -- The derived class may add data elements to the structure or methods to the actions of its parent class. In this way, it can extend the parent class.
2. Override components -- Methods that are inherited from the parent class can be overridden by providing new def-
3. Remove components -- Methods that are defined in the parent class can be removed from the derived class. This removal is a reversal of the inheritance and is not directly provided by all languages as a part of inheritance.

As an example of inheritance, consider the Java class `Bank_Account` shown in Figure 8.13. Here there are four data items, one constructor, and three methods. In Figure 8.14, we define a class `Check_Account` that is derived from `Bank_Account`. In class `Check_Account`, one data element, `monthlyServiceCharge`, has been added. In addition, one method, `check`, has been added, but note that this is just a renaming of the method `withdraw` from its parent class. For objects of class `Check_Account`, `check` and `withdraw` are synonymous. Also, one method, `monthEnd`, has been overridden, taking on a new definition in the derived class.

**Figure 8.13 Bank_Account class in Java**

```java
public class Bank_Account {
    protected double balance = 0.0;
    protected String monthlyReport = "";
    protected String name = null;
    protected String acctID = null;

    public Bank_Account(String name, String acctID) {
        this.name = name;
        this.acctID = acctID;
    }

    public void deposit(String itemID, double amt, String date) {
        balance = balance + amt;
        monthlyReport += "\nDEP " + itemID + amt + date + balance;
    }

    public void monthEnd() {
        System.out.println(monthlyReport);
        monthlyReport = "";
    }

    public boolean withdraw(String itemID, double amt, String date) {
        if (amt > balance)
            return false;
        else {
            balance -= amt;
            monthlyReport += "\nWTH " + itemID + amt + date + balance;
            return true;
        }
    }
}
```

**Figure 8.14 Check_Account class in Java**

```java
public class Check_Account extends Bank_Account {
    double monthlyServiceCharge = 0.0;

    public Check_Account(String name, String acctID, double serviceCharge) {
        super(name, acctID);
        monthlyServiceCharge = serviceCharge;
    }

    public void monthEnd() {
        withdraw("SVC", monthlyServiceCharge, "");
        super.monthEnd();
    }
}
```
public boolean check(String itemID, double amt, String date) {
    return withdraw(itemID, amt, date);
}

Reinforce: In a human family, a person can draw a tree of biological relationships and can thus trace inheritance. Draw such a tree for the classes in Figure 8.14. You will need different relationships to be specified for “extends” and “implements”.

8.4.2 Advantages and Disadvantages of Inheritance

The obvious advantage of inheritance is that it facilitates code reuse. When two classes have significant overlap, inheritance makes that overlap possible without the duplication of large portions of the code or data definition.

In addition, inheritance facilitates the expression in a programming language of the “is-a” relationship between classes. In the preceding example, a Check_Account “is-a” Bank_Account making the inheritance relationship a natural expression of the fact that the properties and actions of a Bank_Account are a subset of the properties and actions of Check_Account.

Inheritance also enables the use of abstract classes, a useful tool in object-oriented system design. An abstract class is a class that is not intended to be instantiated but rather exists to form a set of properties and actions common to a number of classes that are to be instantiated. A conceptual example of such a class would be the hypothetical class mammal. There are no instantiations of the mammal class. It only exists to provide a collection of properties and actions that are common to a number of species, such as human, dog, and cat, all of which are instantiated in the form of specific animals. Note that abstract classes are frequently parent classes for other abstract classes, such as the class dog, which can be considered abstract with various breeds derived from it.

The most prominent disadvantage of inheritance is it detracts from the readability of a class in that some aspects of a class’s definition are implicit and not visible in the definition of the class. To see all of a class definition, you must refer to the definition of another, possibly non-locally defined class, its parent class. In the case where there may be several layers of inheritance or where multiple inheritance is permitted, there may be multiple nonlocal environments that must be referenced.

8.4.3 Multiple Inheritance

One issue that stimulates controversy in the object-oriented world is multiple inheritance, the ability of a class to be the derived class of more than one parent class. There are many situations where multiple inheritance can be useful. For example, a class GraduateStudent might inherit from both the class Student and the class CollegeGraduate.

The controversy surrounding multiple inheritance arises because, in spite of its convenience, it can also be problematic. Multiple inheritance adds to the complexity of a class by the presence of multiple external, implicit definitions. The interactions between these multiple definitions are difficult to understand and may compromise encapsulation. For this reason, the design of some languages omit multiple inheritance, though other languages, such as C++ and Eiffel, do support it. Further information about multiple inheritance in C++ is found in Chapter 11.

8.4.4 Abstract Classes and Inheritance in Java

Many programming languages include features that support the definition of abstract classes, those classes that are not intended to be instantiated but are created so that other classes can inherit from them. This is usually done by attaching the keyword abstract to the definition of the class. Abstract classes are like other classes with two exceptions: 1) they may not be instantiated, and 2) some of their methods may be specified as abstract. An abstract
method is one whose name, number and type of parameters, return type, and throws clause are specified in the class but whose implementation is not. Such methods require that an implementation of the method be given in some other class that is derived from the abstract class where the method is abstract.

The Bank_Account from section 8.4.1 could have been defined as an abstract class. Figure 8.15 shows an abstract version. Here the two methods monthEnd and withdraw are declared abstract. This means that any non-abstract class that is derived from this abstract class must provide an implementation of these two. It is possible to derive a Java abstract class from another abstract class.

Figure 8.15 Class Bank_Account as a Java Abstract Class

```java
public abstract class Bank_Account {
    protected double balance = 0.0;
    protected String monthlyReport = "";
    protected String name = null;
    protected String acctID = null;

    public Bank_Account(String name, String acctID) {
        this.name = name;
        this.acctID = acctID;
    }

    public void deposit(String itemID, double amt, String date) {
        balance = balance + amt;
        monthlyReport += "DEP " + itemID + amt + date + balance;
    }

    public abstract void monthEnd();

    public boolean withdraw(String itemID, double amt, String date);
}
```

8.4.5 Interfaces and Inheritance

Java provides a full abstract class specification called an interface. The Java interface consists of abstract method declarations and constant field declarations. No variable fields or non-abstract methods are permitted.

An interface is then implemented by classes. This is done in the class declaration with the form

```java
class <class_identifier> implements <interface_identifier> {,<interface_identifier>}
```

From this syntax, it is obvious a class may implement one or more interfaces. This implementing class must provide the implementation of all methods from all interfaces it extends. The class also inherits all of the constants that are declared in the interfaces it extends.

It is interesting to examine the relationship between interfaces and inheritance at several levels. The first is that the implementation of an interface is really identical to inheritance from a fully abstract class. In this way, Java provides a limited capability for multiple inheritance as long as all but one of the parents are fully abstract classes in the form of interfaces.

Another way inheritance intersects interfaces is that interfaces themselves may inherit from other interfaces. This means an interface may extend another interface, inheriting its abstract methods and its field constants.

Finally, a class inherits all of the interfaces of its parent class. This has no impact on the derived class, however, since the parent class must already implement its interfaces and that entire implementation will be inherited by the derived class. Therefore, although the derived class inherits its parent’s interfaces, it does not need to implement them explicitly, as the implementation will be inherited.
The example in Figure 8.16 shows this interaction of inheritance and interfaces. Here we have a hierarchy of three interfaces that inherit from each other. The class One_Way_Cycle_Counter implements One_Way_Bounded Counter interface, which includes the inherited interface One_Way_Counter. The class Two_Way_Cycle_Counter implements Two_Way_Bounded Counter and extends One_Way_Cycle_Counter. Through both of these it implicitly implements the other two interfaces.

**Figure 8.16 Interface and Class Hierarchy for Counter in Java**

```java
public interface One_Way_Counter {
    void increment();
    int get_value();
    void set_value(int value);
}

public interface One_Way_Bounded_Counter extends One_Way_Counter {
    void set_bounds(int lower, int upper);
    int get_upper();
    int get_lower();
}

public class One_Way_Cycle_Counter implements One_Way_Bounded_Counter {
    protected int value;
    protected int upper;
    protected int lower;
    public One_Way_Cycle_Counter(int lower, int upper) {
        set_value(lower);
        set_bounds(lower, upper);
    }
    public int get_value() {return value;}
    public int get_upper() {return upper;}
    public int get_lower() {return lower;}
    public void set_value(int v) {value=v;}
    public void set_bounds(int l, int u) {lower=l; upper=u;}
    public void increment() {
        if (get_value() >= get_upper()) set_value(get_lower());
        else value++;
    }
}

public interface Two_Way_Bounded_Counter extends One_Way_Bounded_Counter {
    void decrement();
}

public class Two_Way_Cycle_Counter extends One_Way_Cycle_Counter implements Two_Way_Bounded_Counter {
    public Two_Way_Cycle_Counter (int lower, int upper) {
        super(lower, upper);
    }
    public void decrement() {
        if (get_value() <= get_lower()) set_value(get_upper());
        else value--;
    }
}
```

**Discuss:** What is the difference between inheritance and the use of include files such as in the language C?

**Discuss:** How are the concepts of inheritance and derived types similar and different?

**Discuss:** Some of the difficulties with inheritance -- especially multiple inheritance -- show up in the specification of programs. In languages like Java, these show up often in the complex
inheritance trees the languages use. Discuss ways to get around these problems -- including what development software could do to help a programmer.

Discuss: We discussed that Java uses two keywords -- “extends” and “implements” -- to depict relationships between classes. Describe the difference between these. Why could there not simply be one keyword? Could you interchange them?

8.5 Polymorphism

Polymorphism is the ability of a single representation to have multiple interpretations. As a property found in programming languages, this refers to the ability of the same syntactic structure to take on different semantic meanings in different contexts. We examine two forms of compile time polymorphism we have already seen. We also introduce a run-time polymorphism that is made possible by inheritance.

8.5.1 Polymorphism through Overloading

The first form of polymorphism that we have seen is overloading. In this case, an operator, function, or method takes on different meanings depending on the types of its parameters. For example, in Java the operator + takes on different meaning depending on the type of its operands. The expression

\[ a + b \]

specifies integer addition if \( a \) and \( b \) are an integer type, floating addition if \( a \) and \( b \) are a floating type, and concatenation if \( a \) and \( b \) are of class \text{String}.

In a similar way, methods can be overloaded to take on different meanings depending on their parameter types. For example, suppose class \text{Counter} included two different methods named \text{set_value}:

\[
\begin{align*}
\text{public void set_value(int v) \{} \\
\text{\quad value = v;} \\
\text{\}} \\
\text{\} \\
\text{\text{public void set_value(String s) \{} \\
\text{\quad value = Integer.parseInt();} \\
\text{\}}}
\end{align*}
\]

In this case the statement

\[ \text{aCounter.set_value(x);} \]

takes on different meanings depending on the type of \( x \).

A further form of method overloading differentiates by the class of the receiver of a method call. In Java, the method call

\[ x.toString() \]

specifies different actions for different classes of \( x \), though the result is the same: return a String representation of the object.

In all of the cases of overloading described here, the selection of the method from the syntactic form is per-
formed at compile time, because the type or class of each parameter and receiver is known at compile time. We will see soon that inheritance can make compile-time determination impossible.

8.5.2 Polymorphism and Parameterization

Another form of polymorphism is that provided by type parameterization as discussed in Section 8.3. Consider the binary search tree package given in Figure 8.9. The call

```
insert_in_tree(anItem, aTree)
```

could specify a different action depending on the instantiated type of `aTree`.

This form of polymorphism again can bind the syntactic form to its action at compile time, as the type of `aTree` is known during compilation.

8.5.3 Polymorphism and Inheritance

Inheritance introduces a new form of polymorphism. Let us demonstrate this by a simple example. In Figure 8.17, we define an abstract class `Animal` and two classes derived from this class, `Dog` and `Cat`. Each of these derived classes has its own version of the `speak` method. In the following sequence of statements, the first call of `speak()` will call the `Dog` class’s `speak` method, and the second will call the `Cat`’s

```
Animal a = new Dog();
System.out.println(a.speak());
a = new Cat();
System.out.println(a.speak());
```

In this example, `a` is a variable that references objects of class `Animal` and as such may reference any objects of any class that is derived, either directly or indirectly, from `Animal`. Here `a` is first assigned a reference to a `Dog` and then, later in the execution, a `Cat`. When `a.speak()` is called, a run-time determination is made of the class of `a`, and the appropriate `speak` for that class is called. If a compile-time determination were made, `a` would be considered to be of class `Animal`. In that case, `a.speak()` would call the `speak` method of `Animal`. Since Java makes the binding at run time, the above code give the following output:

```
woof
meow
```

This binding of the method call to the actual method at run time is an excellent example of dynamic binding discussed in Chapter 3.

Figure 8.17 Animal, Dog, and Cat Classes in Java

```java
public class Animal {
    String speak() {return "oops";}
}

public class Dog extends Animal {
    public String speak() {return "woof";}
}

public class Cat extends Animal {
    public String speak() {return "meow";}
}
```
Discuss: To what degree is the concept of polymorphism implemented in Ada?

Discuss: Describe the difference between polymorphism and inheritance.

Discuss: How does the concept of class differ from that of type? How are they the same?

Discuss: Discuss the wisdom of allowing operator symbols to be overloaded in a language specification. For example, should a language be allowed to overload the “<” operator or should the programmer be forced into using a function like “LESS_THAN” or “LESS_THAN_FLOAT” as discussed in this chapter?

Laboratory: 1. Design and implement an abstract data type for rational numbers.

Laboratory: Implement an abstract data type for stacks.

Laboratory: Implement an abstract data type for polynomials. Use parameters if available.

Laboratory: Implement an ADT for class Amount in an imperative language.

Laboratory: Implement an ADT for class Account in an imperative language.
Terms - Chapter 8
encapsulation
parameterization
abstract data type
own variables
instantiation
monitor
package
package specification
package body
private type
own variable
generic package
object-oriented
object
class
message
method
encapsulation
inheritance
multiple inheritance
polymorphism
dynamic binding