Chapter 7 Procedural Abstraction

Among the most powerful of the tools contained in a programming language are those that allow procedural abstraction. An abstraction is a representation of an object that hides what could be considered as irrelevant details of that object and thus makes working with the object easier. Procedural abstraction involves abstracting out the relevant details of a procedure from the irrelevant ones (Liskov and Guttag 1986). The two ways that it does this--through parametric abstraction and through specification abstraction--are described in detail.

The syntax of procedure definition and invocation is examined. The environment in which a procedure operates is studied, with special attention paid to scope and binding of data objects. Languages offer a variety of methods for passing parameters, and we describe and compare these variations. Whereas the common form for procedures serves as an abstraction of a statement, the value-returning procedure is an abstraction for an expression.

Some languages permit a further abstraction of procedures over parameter type through a technique called overloading. The concept of coroutines is introduced, and the procedure constructs found in the programming language Java are summarized.

We examine the language feature known as exception handling, where an abstract block of statements is invoked implicitly by the occurrence of a condition rather than explicitly by a call to a procedure and describe the approach used for exception handling in the programming language Java.

7.1 Procedures as Abstractions

We have seen examples of abstraction in our earlier study. The record type, for example, is a low-level abstraction. This type permits the user to manipulate a record by name, ignoring the details such as component names and types when such details are irrelevant. Such an abstraction is useful when one wishes to assign one entire record to another or pass an entire record as a parameter.

In this chapter we consider the procedure as an abstraction of a program unit into a simpler execution unit such as a statement or an expression. As with all abstractions, irrelevant details will be hidden from the user of the abstraction.

The advantages of using procedural abstractions are as follows:

1. Program units are simpler. This simplicity results in units that are easier to read, write, and modify. By hiding lower levels of detail, a unit can focus on a single task. This property is important in the implementation of top-down design.
2. Program units are independent. Abstraction permits the actions of a procedure to be independent from its use. Therefore, the program using the abstraction is not affected by the details of the abstraction's implementation.
3. Program units are reusable. A procedure, once defined, can be used within many different programming environments. This eliminates redundant programming effort and reduces errors.
In order to understand the role of procedural abstraction, recall the four levels of execution units described in Chapter 4. These execution units are expression, statement, block, and program, listed in increasing order of complexity. A statement may contain several expressions, a block may contain several statements, and a program may contain several blocks. Procedural abstraction is, in effect, the representation of one execution unit by another that is simpler. In practice, it is commonly the representation of a block by either an expression or a statement.

As an example, consider the Java method below:

```java
void USELESS() {
    System.out.println("This is the result");
    System.out.println("of an execution");
    System.out.println("of method USELESS.");
}
```

This defines `USELESS` to be a block of three statements that can be represented as a single statement, namely,

```java
USELESS();
```

This abstracted statement can be used anywhere that a statement is appropriate. Similarly, a Java method that returns a value defines a block that can be represented as an expression.

Procedure types and procedure objects have various bindings, which are analogous to the bindings of data types and data object defined in Chapter 4. Figure 7.1 illustrates the bindings of a procedure type. The procedure type is bound to its name and its executable block by its declaration. These bindings both occur at compile time. Further bindings are introduced in later sections as they are required.

**Figure 7.1 Procedure Type**

---

7.2 Procedure definition and invocation

The definition of the procedure specifies all the compile-time bindings that must occur. It is analogous to the definition of a data type. At this point, the two bindings that must be specified
are name and executable block. For example, the simple form for Java is

```java
void <procedure_name>()
{
    <procedure_block>
}
```

The invocation of a procedure is written as the procedure name, and it represents an abstract statement. This causes the executable block of statements associated with the procedure to be executed. In effect, it results in a further binding, the binding of the procedure object to an activation of the procedure. This binding, by its very nature, must occur at run time. A single defined procedure may have any number of activations at a given time. The activations of a procedure are similar to the variables of a given data type. Whereas the entrance of a block containing a declaration of a data variable creates a data object bound to the declared type, the invocation of a procedure creates the procedure object bound to the named procedure definition. This binding is illustrated in Figure 7.2.

**Figure 7.2 Procedure Activation**

This analogy between data types and procedures is very important. Consider the following examples. The statement

```
typedef ... T;
```

declares T to be a data type that is bound to its definition at compile time. The statement

```
T V1,V2;
```

declares V1 and V2 to be variables, or instances, of type T that are bound to their types and locations at run time. In this case, there may be many instances of the same type, each instance bound at run time.

Similarly, the declaration
void T() {
    ...
}

declares T to be a procedure that is bound to its definition at compile time. Each invocation of T of the form

T();

causes an activation of T that is created at run time. Each activation binds the procedure type T to an activation record in the same way a variable declaration binds the data object to its data type.

### 7.3 Procedure environment

#### 7.3.1 Activation record

The invocation of a procedure results in the creation of an **activation record** for that procedure. The general form of the activation record consists of three parts: the local environment, the parameter environment, and the nonlocal environment reference. The local environment contains all data objects that are defined locally inside this procedure. The parameter environment contains information about all data objects that are passed to and from the procedure. We postpone a detailed discussion of this environment until Section 7.4. The nonlocal environment reference points to the activation record from which the present activation will inherit objects and their bindings. We will call such inherited objects **nonlocal objects**.

When a procedure is invoked, its activation record is pushed onto the **run-time stack** of activation records. When the procedure terminates, its activation record is popped off the run-time stack, making its invoking program unit the new top-of-stack activation record and hence the currently active program unit. Therefore, the run-time stack contains the activation records of all program units that are currently in the midst of execution. Only the unit whose activation record is on top of the stack is actively executing. The other units are suspended until all units above them in the stack are terminated and thus popped off of the stack.

To illustrate the use of activation records and the run-time stack, consider the Java program shown in Figure 7.3. The state of the run-time stack at each stage of execution is also displayed in this figure. Note that the invocation of a procedure causes an activation record to be placed on top of the stack; its termination pops off that activation record. Also note that recursion, as with method r, results in the stacking of multiple activation records of the same method. In this figure, no details are shown for the contents of the activation record. These are filled in later.
7.3.2 Local environment

The local environment of a procedure includes the locally declared data objects. They are specified in the local environment section of the activation record. Also included in the local environment is a pointer to the next instruction to be executed in the procedure, known as the return address pointer. This pointer permits the activation record to store the location where execution will resume upon return from an invocation, and allows multiple activations of the same procedure to specify different execution addresses.

One further category of information in the local environment is the temporary storage needed to retain data within expression evaluations. For example, execution of a procedure may be suspended by a function call in the middle of a statement. The parts of the expression that have already been evaluated must be stored in the local environment so they can be accessed when the invoked function returns control. This temporary storage is highly implementation dependent, so
we do not mention it any further in our discussion.

If we extend the example of Figure 7.3 to add some local data objects and to show the contents of the local environment part of the activation record, we have the resulting Figure 7.4. The arrows in this figure represent the pointers to the next executable statement.

**Figure 7.4 Sample program and run-time stack**

```cpp
class test {
  int i;

  void q() {
    int vq;
    ...
  }

  void r() {
    int vr;
    i = i - 1;
    vr = i;
    if (vr > 0)
      r();
    else
      q();
  }

  void s() {
    int vs;
    r();
    q();
  }

  void p() {
    int vp;
    i = 2;
    s();
  }
}
```

### 7.3.3 Nonlocal environment

Many programming languages permit a procedure to access data objects other than those local to the procedure itself. The environment where these objects are defined and bound is called the procedure's **nonlocal environment**. This environment can be represented in the procedure's activation record by a pointer to the activation record whose local and nonlocal environments con-
stitute the nonlocal environment of the present activation record. In other words, when a data object is referenced whose name is not bound in the local environment, the nonlocal environment is searched to satisfy the reference.

The most common manner for defining the nonlocal environment is through static scope. In this case, the nonlocal environment of a procedure is inherited from the program unit in which the procedure is defined. Stated differently, any name used in a procedure but not bound within that procedure inherits the binding so that name in the program unit immediately containing the procedure. Note that this is identical to the static scope definitions of blocks we introduced in Chapter 3.

We illustrate this static definition of nonlocal environments through the example in Figure 7.5. In this case, we use Ada syntax, because Java does not permit procedure definitions within other procedure definitions. These bindings are completely specified at compile time since they are determined by the placement of the procedures within the program itself. The procedures themselves are bound to their names in the same way as variable bindings. Note, for example, that procedure \( r \) in Figure 7.5 is not a part of the nonlocal environment of procedure \( s \), because it is not bound in \( s \) or in \( p \), the containing unit of \( s \).

**Figure 7.5 Example to illustrate static definition of global environment**

```ada
program p;
var a,b,c : integer;

procedure q;
var a,c : integer;
procedure r;
var a : integer;
begin {r}
    {variables: a from r; b from p; c from q
    procedures: q from p; r from q}
end; {r}
begin {q}
    {variables: a from q; b from p; c from q
    procedures: q from p; r from q}
end; {q}

procedure s;
var b : integer;
begin {s}
    {variables: a from p; b from s; c from p
    procedures: q from p; s from p}
end; {s}

begin {p}
    {variables: a,b,c from p
    procedures: q,s from p}
end. {p}
```
Although the nonlocal environment of a procedure can be determined at compile time, it must be specified at run time in the activation record, because the procedure must refer to a specific activation record of the containing unit. Suppose the run-time stack for the program of Figure 7.5 has been constructed as in Figure 7.6. Here we specify only the nonlocal environment reference. Each activation record points to the most recent activation record of the containing program unit. Notice the importance of the phrase *most recent*. In the case of \( r \) in Figure 7.6, the containing unit is \( q \), but \( q \) has two activation records on the run-time stack. The question is, to which version of \( q \) should \( r \) refer when accessing variable \( c \)? This is resolved by specifying the most...
recently invoked version, that is, the one whose activation record is nearest the top of the stack. The nonlocal environment references now form a linked list that specifies the nonlocal environment of a procedure. In Figure 7.6, the chain beginning at \( \times \) goes through \( q \) and \( p \). This means that if a reference is unresolved in \( \times \), the second activation of procedure \( q \) will next be interrogated, and if the reference is still unresolved, \( p \) will be tried. This chain represents the nesting of procedures within other program units in the physical code of the program.

**Figure 7.7 Nonlocal environment - dynamic scope pointers**

A second natural approach to the binding of the nonlocal environment to a procedure is to use the environment of the program unit that invokes the procedure as that procedure's nonlocal
environment. This technique is called **dynamic scope**, and under the dynamic scope rule, the run-time stack in Figure 7.6 would be modified as shown in Figure 7.7. Note that dynamic scope means a procedure inherits as its nonlocal environment the environment of the program unit that invokes it and hence can be determined only at run time. On the other hand, with static scope, the nonlocal environment is inherited from the program unit that physically contains the procedure. This inheritance relationship that can be determined at compile time except for the case of multiple executions of that containing procedure as described earlier. Note that although the illustrations in Figures 7.6 and 7.7 involve nested procedure calls, static and dynamic scope can cause different actions even when such nesting does not occur.

Upon initial observation, dynamic scoping seems to have several advantages over static scope. First, the nonlocal variable reference could actually be eliminated altogether with dynamic scope, because the chain of activation records follows the physical ordering of the stack. In other words, the invoking unit is always the next one down the stack.

**Figure 7.8 Example illustrating dynamic scope**

```pascal
program p;
var a : integer

procedure q;
begin
    {variables: a from p or r
    procedures: q,r from p}
end; {q}

procedure r;
var a : integer;
begin
    {variables: a from r
    procedures: q,r from p}
    q;
end; {r}

begin {p}
    {variables: a from p
    procedures: q,r from p}
    q;
end. {p}
```

However, the dynamic nature of the environment proves to be a severe detriment to writing understandable programs, since the nonlocal environment of a procedure cannot be determined by examining the source code. Consider, for example, the program in Figure 7.8. Assuming dynamic scope, a reference to variable a in procedure q could refer to variable a from p or variable a from r depending on the point from which q is invoked. This situation greatly detracts
from the understandability of the program and could be the source of great confusion. For this reason, static scope is the chosen method of defining a nonlocal procedure environment in most programming languages.

A third approach to nonlocal environments also has some merit. This is disallowing them altogether, requiring the entire environment to be local or passed explicitly. The use of nonlocal environments is frequently discouraged, because it produces side effects. One side effect is the changing of the environment of a program unit without the change being specified in the code of that unit. An invoked procedure modifying a variable it inherits from its nonlocal environment is an example of this. Java adopts this approach by not allowing the nesting of program units. This means a Java method is never contained within another method. All Java methods are contained within a class, however, and member data of that class can be modified inside a method of the class, producing a localized side effect.

Discuss: State some advantages of dynamic scope over static scope in defining the nonlocal environment of a procedure.

Discuss: Give the relative advantages and disadvantages of disallowing nonlocal environments in procedures.

Discuss: It was suggested that nonlocal data objects be declared in a containing program unit. An extension of this would be to specify in the declaration the name of the program unit whose binding is to be used for every nonlocal object. How might this look syntactically? What are some difficulties with this approach?

Discuss: Some languages provide multiple entry points into the same procedure that are called by different names and may have different parameters associated with them. What are the benefits and dangers of such a feature?

Reinforce:

program Exercise2 (input, output);
  var a,b: integer;
  c,d: real;
procedure p1;
  var d: real;
    x: boolean;
  begin
    {****** HERE!!!! ******}
  end;
procedure p2;
  var b,c: real;
procedure p3;
  var x,b: real;
  begin
    a := 1; p2; x := a; p1
  end;
begin
  if a=0 then p3 else p1
end;
begin
  a := 0; p2; p1;
end.

At the point marked {****** HERE!!!! ******}, give the activation record structure of the run-time environment the first time p1 is called. Note that the calling sequence is

\[ \text{main} \rightarrow p2 \rightarrow p3 \rightarrow p2 \rightarrow p1 \]

### 7.4 Parameters

The third part of the procedure environment is the parameter part. **Parameters** are the means by which information is passed between the invoking and called units. We categorize the parameters in two ways: (1) **value** parameters, which pass the actual value of the parameter into the procedure, and (2) **reference** parameters, which pass the reference or address of the parameter into the procedure.

A parameter needs to be specified at two points: in the invoking unit and in the procedure definition. The specification in the invoking unit is called the **actual parameter**. It specifies what is to be communicated to the procedure. For a value parameter, this actual parameter can be any valid expression of the parameter’s type. For a reference parameter, the actual parameter must be a specification of the location of an object of a compatible type, such as a variable name.

A **formal parameter** is the specification of the parameter in the procedure’s definition and is an identifier that is of the specified type.

#### 7.4.1 Parameter association

In a procedure, the formal parameter is represented by a name that is referenced just like a variable within the procedure body. It is also customary for the type of the parameter to be specified in the procedure header by attaching a type to the formal parameter. Corresponding actual and formal parameters are required to be of the same type in a strongly typed language. Some implicit conversion might occur in languages that are not strongly typed. These bindings occur at compile time.

There are two modifications to this requirement that allow the formal parameter to be bound to a type at run time when the procedure is called. The more radical approach is to leave the formal parameter untyped and have it take on the type of the actual parameter whenever an invocation occurs. This is not compatible with compile-time type checking, however, since the type of the formal parameter in this situation is unknown at compile time. A less severe alternative is to permit an unconstrained array type for the formal parameter. The formal parameter then takes on
the index constraints of the corresponding actual parameter at the time of invocation. This permits compile-time type checking to prevail, because the base type or range of the formal parameter array is specified in the procedure heading. Overloading and generics are other facilities for dealing with type correspondence and they will be discussed later.

There are two possible methods for associating actual and formal parameters. They are by position and by name. **Positional parameter association** is used by most programming languages and simply associates the actual and formal parameters according to their relative positions in their parameter lists. **Named parameter association** requires that the name of the formal parameter be appended to the actual parameter in the invocation statement. Suppose that the following is the heading of an Ada procedure:

```ada
procedure TEST(A:in Atype; B:in out Btype; C out Ctype)
```

Then a call of TEST using positional association might be

```ada
TEST(X,Y,Z);
```

Here the actual parameters X, Y, and Z would be associated with formal parameters A, B, and C, respectively. A named association call is

```ada
TEST(A=>X, C=>Z, B=>Y);
```

In this case, the formal parameter name is attached to the actual parameter to indicate association. Finally, Ada permits a mixture of the two, such as

```ada
TEST(X, C=>Z, B=>Y);
```

In the mixed association, positional can be used for all parameters up to the first named association, after which all remaining associations must be named.

One further association technique is the **default parameter association**. This permits the specification of default values for formal parameters in the procedure header. When an invocation is made with no actual parameter associated with the formal parameter, the default value is used. This is, of course, only appropriate for value parameters.

Discuss: When procedure parameters can be associated by name as in Ada, some would argue that all parameters should be associated in this way since it provides greater clarity than the positional association. Do you agree?

### 7.4.2 Aliasing

One problem that can arise in the passing of reference parameters and the use of nonlocal environments is **aliasing**. Aliasing is the ability to reference the same location by different names.
To illustrate this problem, consider the C++ program in Figure 7.9. Here the assignment statement in the function uses three data objects, \(X\), \(A\), and \(Y\), where \(X\) and \(Y\) are formal parameters and \(A\) is a nonlocal variable. The call statement \(\text{TEST}(A, A)\) causes all three data objects to be bound to the same location, hence creating aliasing. This practice drives the understandability of the procedure to new lows and forces the programmer to depend on deeper levels of the language implementation. The program prints

\[
2 \ 2 \ 2
\]

**Figure 7.9 C++ program to illustrate aliasing**

```
int A;

void TEST(int& X, int& Y)
{
    X = A + Y;
    cout << A << X << Y;
}

public static void main(String args[]) {
    A = 1;
    TEST(A,A);
}
```

Aliasing can occur when nonlocal variables and formal reference parameters share the same location or when two actual parameters share the same location. By their definition value parameters eliminate the possibility of aliasing.

Some languages, such as Euclid, prevent aliasing by detecting situations that lead to it at compile time and, where that is impossible, generating run-time checks to detect its occurrence. Such testing generates a significant amount of overhead, usually too much for practical use. Languages that do not permit reference parameters or nonlocal environments also avoid aliasing.

*Reinforce: Consider the code below:*

```
program Exercise1 (input, output);
    var a,b,c: integer;

    procedure p1 ([MODE] a,b: integer);
    begin
        a := a * b;
        if (c/b)=a then a:=0 else a:=100;
    end;

    procedure p2 ([MODE] a,b: integer);
```
begin
  a := a - b;
  if a=c then p1(b,a) else p1(a,b);
end;

begin
  a := 1; b := 5; c := 10;
p2(c,b);
end.

**Give the values of** \( a \), \( b \), and \( c \) **from the main program after execution when each of two parameter passing modes are used for** \([\text{MODE}]\) **in Exercise 1 above. As a reminder, the parameter passing modes are:**

- **value**
- **reference**

**Reinforce: Consider the program below:**

```
program Exercise4 (input, output);
var a,b,c: integer;
d: boolean;
procedure p1(var q:boolean; var r, s:integer);
begin
  if d then r := 100 else r := 200;
s := s / a;
end;
procedure p2(var x,y: integer; var z: boolean);
begin
  x := 15; y := x + a; z := (x<a);
p1(z,y,x);
z := (x<a);
end;
begin
  a := -1; b := -1; c := -1; d := true;
p2(a,b,d);
end.
```

**Call by Value-Result** is a parameter passing scheme whereby the value of the actual parameter is passed to the formal parameter before execution of the procedure, just as for value parameters. When the procedure terminates normally, the values of the formal parameters, which may have been modified during the procedure execution, are passed back to the actual parameters.

Using call by value-result semantics as the parameter passing mode, what would the final values of the variables in the main program be at program termination?
7.4.3 Procedures as parameters

An additional feature available in some languages is the use of procedures as parameters. In specifying the formal procedure parameter, it is necessary that the types of all parameters to the procedure itself be specified as well, so type checking can be performed when the formal procedure is called. Figure 7.10 shows a Pascal program that uses a procedure as a parameter. Procedure TESTPOS is called twice, the first time using E1 as the actual parameter procedure for formal parameter ERROR, and the second time using E2 as the actual parameter.

One confusing aspect of the Pascal implementation is that the formal procedure parameter must have formal parameter names expressed in its definition, whereas only the types of these parameters are used. This formal parameter name is MSG in Figure 7.10. Such formal parameters of formal procedure parameters have no meaning within the procedure but appear only as place holders so the parameter types can be specified.

A further consideration in the use of procedures as parameters is the information about them that must be passed into a procedure at invocation time. There are two items that must be passed into a procedure: the location of the executable code and a pointer to the nonlocal environment's activation record. These two items are all that are needed to create the activation record for the parametric procedure when it is invoked.
7.4.4 Name parameters

A different form of parameter passing was implemented in the ALGOL 60 language. This is called the name parameter or parameters passed by name. Such parameters are implemented by the binding of the name of the actual parameter to the formal parameter on invocation of the procedure. This is conveniently visualized as a run-time textual substitution of the actual parameter name for the formal parameter within the procedure.

Name parameters behave identically to reference parameters implemented by reference as long as the name of the actual parameter remains bound to the same location throughout the execution of the procedure. This may not be the case, however, especially when an actual parameter is an array element. Consider the procedure

```pascal
procedure swap(a,b: integer);
var temp: integer;
begin
  temp := a;
```
a := b;
b := temp;
end;

If we assume the formal parameters \( a \) and \( b \) are name parameters, then in the program

```pascal
program main;
var i:integer;
    m:array [1..100] of integer;
...
begin
...
    swap(i,m[i]);
...
end.
```

the call to `swap` will yield a different result than if the parameters were reference parameters. Since the names are \( m[i] \) and \( i \) are passed, the procedure, after textual substitution, will be executed as

```pascal
temp := i;
i := m[i];
m[i] := temp;
```

The index \( i \) in the third line will be modified by the assignment in the second line, resulting in the destination location of the final assignment being different from the location referenced on the right-hand side of the first assignment.

Another anomaly occurs with name parameters when one of the actual parameter's names is the same as a local variable of the procedure. Suppose our main program which calls `swap` is

```pascal
program main;
var i,temp:integer;
...
begin
...
    swap(i,temp);
...
end.
```

Name parameter passing will result in the execution of

```pascal
temp := temp;
i := temp;
```
temp := i;

Confusion results, because there are two data objects bound to the name temp in this procedure, one local to the procedure and the other local to the calling program unit. Each reference to temp in the execution of swap must be resolved to one of these objects. The method used by ALGOL 60 is to bind formal name parameters to the data object to which the corresponding actual parameter name is bound in the invoking program unit. In the case of our example, the two objects named temp would be referenced according to

\[
\begin{align*}
\text{temp}_{\text{swap}} & := \text{temp}_{\text{main}}; \\
i & := \text{temp}_{\text{swap}}; \\
\text{temp}_{\text{main}} & := i;
\end{align*}
\]

Here the subscripts are used to differentiate the references to the two objects bound to the name temp.

Because of the problems just described and the difficulty in implementing name parameters, they have not been provided in programming languages since ALGOL 60.

**Discuss:** One type that can be passed as a parameter that was ignored in this section is the label. Discuss some of the difficulties that might arise if label passing is permitted.

**Reinforce:** Consider the Pascal code below:

```pascal
program Exercise3 (input, output);  
var limit: integer; 
function Summation ([MODE] lim: integer): integer;  
var s: integer; 
begin 
    s := 0; 
    while lim > 0 do 
        begin 
            s := s + limit; 
            limit := limit - 1; 
        end; 
    Summation := s 
end; 
begin 
    limit := 6; 
    writeln('Sum of 1 through 6 is ', Summation(limit)) 
end.
```

Which parameter mode would cause the function Summation to have an infinite loop? Explain your answer.
Reinforce: Write a program that would be described by the activation record structure in Figure 7.21. Indicate where the execution would have to be suspended to produce the structure. The following is the format of the activation records:

- local vars
- parameters
- return addr
- static pointer
- dynamic pointer

It is assumed that the static and dynamic pointers point to the containing modules according to the static and dynamic scope rules respectively.

7.5 Value returning procedures

The second level of procedural abstraction is implemented by value-returning procedures (VRPs) which are commonly known as functions. These represent blocks that are abstractions of expressions and return a single value of their defined type.

In the definition of a VRP, one additional piece of information needs specification: the type of the value returned. Some languages label such procedures as functions, and others label them as procedures and differentiate them from statement abstractions by appending a type to be returned. Another common strategy is to consider all procedural abstractions to be VRPs, with void as the return type for those with no return value. In Pascal, the definition of a VRP is written as

```pascal
function F(X:XType; Y:YType) : ResultType;
```

whereas Modula-2 defines the same VRP with

```pascal
procedure F(X:XType; Y:YType) : ResultType;
```

Other than the use of the words function and procedure, the two definitions are identical. The Modula-2 form emphasizes that a VRP is nothing more than a procedure which returns a value. The third strategy, using a void return value, is used by C++ and Java. When a VRP is invoked, it is placed into the same context in the invoking program that any expression of the same type might occupy.

Within the VRP, there are two common methods employed to specify the value that is to be returned. The first, used by Pascal, is to create a pseudovariable in the local environment that is bound to the name of the VRP. A pseudovariable differs from a variable in the sense that it is not declared within the VRP and it can only be modified and not accessed within the VRP. It can never appear as a variable on the right-hand side of an assignment statement, for example. The
value stored in that variable on termination of the VRP is the value returned to the invoking program. This method requires that a data object for the return value be added to the local environment part of the activation record of the VRP.

The second approach, found in Java, is to use a return statement: an expression following the word return. When this statement is executed, the expression is evaluated, the VRP is terminated, and the value of the expression is returned as the value of the VRP.

One further consideration with VRPs is the mode permitted for the parameters. There is no difficulty in implementing the same two modes (value and reference) that are used in standard procedures. The side effects of using reference parameters in VRPs are not desirable, however, because an expression, which the VRP abstracts, should not change any values in the environment.

**Figure 7.11 Example of overloading in Java**

```java
int F(double x)
{ return 1; }

int F(int x)
{ return 2; }

int F(double x, int y)
{ return 3; }

int F(int x, double y)
{ return 4; }

public static void main(String args[])
{
    double d = 0.0;
    int i = 0;
    System.out.println(F(d));
    System.out.println(F(i));
    System.out.println(F(d,i));
    System.out.println(F(i,d));
}
```

### 7.6 Overloading

**Overloading** of procedures permits two or more procedures to have the same name if they can be distinguished by the number or type of their parameters or, in the case of a VRP, by the type of the return value. Such overloading permits procedures that perform the same operation on different parameter types to be called by the same name. The use of this feature will be discussed further in Chapter 8 under data abstraction.

The language Java provides a good example of procedure overloading. Figure 7.11 shows
an example of the use of this facility. Here the VRP $F$ is overloaded. The four different definitions of $F$ are distinguished by the types of the parameters, the number of parameters, the order of the parameters, and the type of the returned value. Java does not allow overloading based on the return type.

7.7 Coroutines

A variation of the procedure that is implemented in some programming languages is the coroutine. Whereas a procedure, when invoked, executes from the beginning of its body to the end, a coroutine executes from the point where it last suspended execution up to the next instruction that suspends its execution. In other words, a procedure, when invoked, executes until its entire body is completed, whereas a coroutine may have only a portion of its body executed before it is suspended. The relationship between a coroutine and its invoking body is illustrated in Figure 7.12.

**Figure 7.12 Coroutine execution**

The pseudolanguage model we will use in our description considers coroutines to be variables to which an executable body can be bound. In this way, several coroutine variables may be bound to the same executable body at the same time and therefore be in the midst of execution. All except possibly one will be suspended at any time, because only one coroutine alternative can be actively executing. Also, for our model, we assume that the bodies to which coroutines are bound are described as parameterless procedures. We specify the declaration of coroutine vari-
ables through statements of the form

\[ C : \text{COROUTINE}; \]

Four basic coroutine operations are included in our model. The first is CREATE, which binds the coroutine variable to a parameterless procedure body. Its general form is

\[ \text{CREATE <coroutine_variable_name> FROM <parameterless_procedure_name>} \]

The CREATE statement creates an activation record for the coroutine and binds it to its name and procedure body. It also sets the current execution location in the activation record to the beginning of the procedure.

The DELETE statement deletes the activation record associated with its coroutine variable parameter. Its general form is

\[ \text{DELETE <coroutine_variable_name>} \]

A coroutine is invoked through the RESUME statement, which is of the form

\[ \text{RESUME <coroutine_variable_name>} \]

This sets the invoking unit pointer in the called coroutine to point to the invoking unit and begins execution of the coroutine at its present execution location.

A coroutine in execution may RESUME another coroutine or may return to the point from which it was invoked. The latter action is accomplished through an EXIT statement of the form

\[ \text{EXIT} \]

This suspends execution of the present coroutine and returns to the invoking unit at the statement immediately following the invocation.

The implementation of coroutines requires some modification to our previous stack model of activation records. Because coroutines do not follow the procedure-call protocol, their activation records can be deleted in any order, independent of the order in which they were created. Therefore, the activation records of all coroutines defined in a program unit will form a linked list rather than a stack. Furthermore, the invoking unit of a coroutine must be explicitly identified within the coroutine's activation record. Coroutines will inherit as their nonlocal environment the environment of the activation record in which they are defined. Therefore, the activation record of a coroutine must include the following fields in addition to its local environment:

- Name of coroutine
- Pointer to current execution location of coroutine
- Pointer to invoking activation record
- Pointer to defining activation record
Furthermore, the activation record of the invoking unit must contain a pointer to the list of coroutines the unit has defined. Figure 7.13 illustrates this implementation model through a detailed example. In this figure, execution unit A creates two coroutines, C1 and C2, from the procedures P1 and P2. For each step of the execution, a box is provided that displays the pertinent code of each of the three program modules, and a box representing each activation record is located beneath its associated module. Those components of the activation record that are current instruction pointers always point to the instruction in the preceding code. Similarly, pointers to activation records point to the appropriate activation record boxes.

Figure 7.14 shows an example of a coroutine written in the language ACL, a coroutine language designed by Marlin (1980). ACL is a derivative of Pascal that adds two features: (1) coroutines and (2) explicit scope rules. The ACL code in Figure 7.14 is written in a mutation of the language where the explicit scope rules have been ignored in order to illustrate the coroutine concept.

In this program, the code between initbegin and initend is executed when the create Count statement causes an instance of Count to be allocated. The return statement in Count causes a return from the coroutine, but on the next call to Count, execution begins at the statement immediately following the return statement. Figure 7.15 shows the output generated by the program in Figure 7.14.
Figure 7.13 Illustration of coroutine activation records

Execution Unit A;
... C1,C2 : COROUTINE BEGIN ...
CREATE C1 FROM P1;
CREATE C2 FROM P2;
...
RESUME C1;
...
DELETE C1;
RESUME C2;
DELETE C2;
END A;

PROCEDURE P1;
...
BEGIN ...
RESUME C2;
...
EXIT;
END P1;

PROCEDURE P2;
...
BEGIN ...
RESUME C1 ... EXIT;
END P1;

A
Local Environment
Nonlocal Environment
Current Instruction
Coroutine List

C1
Current Instruction
Invoking A.R.
Defining A.R.
Next in List

C2
Current Instruction
Invoking A.R.
Defining A.R.
Next in List

Execution Unit A;
... C1,C2 : COROUTINE BEGIN ...
CREATE C1 FROM P1;
CREATE C2 FROM P2;
...
RESUME C1;
...
DELETE C1;
RESUME C2;
DELETE C2;
END A;

PROCEDURE P1;
...
BEGIN ...
RESUME C2;
...
EXIT;
END P1;

PROCEDURE P2;
...
BEGIN ...
RESUME C1 ... EXIT;
END P1;

A
Local Environment
Nonlocal Environment
Current Instruction
Coroutine List

C1
Current Instruction
Invoking A.R.
Defining A.R.
Next in List

C2
Current Instruction
Invoking A.R.
Defining A.R.
Next in List

Execution Unit A;
... C1,C2 : COROUTINE BEGIN ...
CREATE C1 FROM P1;
CREATE C2 FROM P2;
...
RESUME C1;
...
DELETE C1;
RESUME C2;
DELETE C2;
END A;

PROCEDURE P1;
...
BEGIN ...
RESUME C2;
...
EXIT;
END P1;

PROCEDURE P2;
...
BEGIN ...
RESUME C1 ... EXIT;
END P1;

A
Local Environment
Nonlocal Environment
Current Instruction
Coroutine List

C1
Current Instruction
Invoking A.R.
Defining A.R.
Next in List

C2
Current Instruction
Invoking A.R.
Defining A.R.
Next in List
Figure 7.13 (continued)

Execution Unit A;
...  
C1,C2 : COROUTINE
BEGIN
...  
CREATE C1 FROM P1;
CREATE C2 FROM P2;
...  
RESUME C1;
...  
DELETE C1;
RESUME C2;
DELETE C2;
END A;

PROCEDURE P1;
...  
BEGIN
...  
RESUME C2;
...  
EXIT;
END P1;

PROCEDURE P2;
...  
BEGIN
...  
RESUME C1
...  
EXIT;
END P1;

PROCEDURE P1;
...  
BEGIN
...  
RESUME C2;
...  
EXIT;
END P1;

PROCEDURE P2;
...  
BEGIN
...  
RESUME C1
...  
EXIT;
END P1;

A
Local Environment  
Nonlocal Environment  
Current Instruction  
Coroutine List

C1
Local Environment  
Nonlocal Environment  
Current Instruction  
Coroutine List

C2
Local Environment  
Nonlocal Environment  
Current Instruction  
Coroutine List

C2
Local Environment  
Nonlocal Environment  
Current Instruction  
Coroutine List

PROCEDURE P2;
...  
BEGIN
...  
RESUME C1
...  
EXIT;
END P1;
PROCEDURE P2;
... RESUME C1
END P2;

EXECUTION UNIT A;
... CREATE C1 FROM P1;
CREATE C2 FROM P2;
... RESUME C1;
... DELETE C1;
RESUME C2;
DELETE C2;

END A;
Laboratory: The Sieve or Eratosthenes is an interesting algorithm to compute prime numbers. It is an example of an inherently concurrent problem, but it is adaptable to coroutines. Suppose we have a set of activations that all do the following:

```plaintext
program Counter (input, output);

coroutine Count;
    var i, counter : integer;
    init
        begin {started only when instance is allocated}
            counter := 0;
        initend;
    begin {started any time a call is made}
        for i := 1 to 5 do begin
            counter := counter + 1; writeln(counter:1);
        end;
        return;
        for i := 1 to 5 do begin
            counter := counter + 1; writeln(counter:1);
        end;
        return;
    end;

var C : instance of Count;

begin
    C := create Count;
    call C;
    writeln(‘Halfway through’);
    call C;
    delete C;
end.
```

Figure 7.15 Output generated by ACL program in Figure 7.14

1
2
3
4
5
Halfway through
6
7
8
9
10
a. Read a number from an input stream.
b. Print this first number.
c. Activate a new routine and pass that activation all numbers from the input stream that are not divisible by the first number.

Each activation becomes a filter. If we have a main process that generates all integers (in some range), activates a filter, and passes this stream of integers on to this filter, we get the configuration found in Figure 7.22. The first number in each filter is prime; the rest are filtered out or passed on.

This makes for an interesting program, because it features (1) an inherently non-procedural problem and (2) an elegant solution. You are to design a solution to the Sieve or Erastothenes in pseudo-ACL. Design and specify the solution using coroutine semantics as we have sketched them in this chapter.

Figure 7.22 Illustration of Filters in Lab Exercise 6

<table>
<thead>
<tr>
<th>Main</th>
<th>2,3,4,5,6,...</th>
<th>filter(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3,5,7,9,...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>filter(3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5,7,11,13,...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>filter(5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7,11,13,...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

7.8 Procedures in Java

Java procedures occur as methods within the object-oriented framework. All methods have a return value, with \texttt{void} used in the case where no value is actually returned from the method.
Parameters of primitive types are value parameters. Parameters of reference types (including arrays and classes) are reference parameters. Method invocation must specify the object or, in the case of static methods, the class that is the recipient of the message. The format of the invocation is

\[
\text{<receiver>.<method_name>(<parameter_list>)}
\]

### 7.9 Exceptions

An **exception** is a condition that requires some immediate action on the part of the program. This condition might be an error, such as arithmetic overflow or index out of range, or it may be an abnormal or rarely occurring condition that is not considered an error. An example of such a condition is end-of-file. An additional type of exception is the modification of a specified storage location. This latter type of exception is useful for the purpose of debugging.

Conditions that are considered exceptions are said to be **synchronous**, meaning that they arise at predictable places in the program. For example, overflow will occur only when arithmetic is being done, index out of range will occur only when an array is being accessed, and end-of-file will occur only when a file is being read. This distinguishes exceptions from conditions that are **asynchronous**, that is, conditions that may occur at any time. A user-generated interrupt or a device-ready signal are examples of asynchronous conditions. These asynchronous conditions are more appropriately handled by the concurrency features of a language, since they are generated by some concurrently executing process.

**Figure 7.16 Procedure Execution**

![Procedure Execution Diagram](Diagram)
Procedures, as we have described them, are invoked through an explicit call that invokes the procedure into execution and suspends execution of the invoking unit. This is illustrated in Figure 7.16. On completion of the procedure, the invoking unit is resumed from the point of invocation.
Figure 7.17 Exception Handling

(a) Resume calling unit

(b) Terminate calling unit

Completion of Execution

Implicit Call

Completed of Execution of exception handler
Procedures invoked implicitly by the occurrence of an exception are called exception handlers. As with a procedure call, exception handling results in suspension of the execution of the invoking unit. Two different actions are possible on termination of the exception handler: resumption of the invoking unit as with procedures or termination of the invoking unit. These two approaches are illustrated in Figures 7.17 (a) and (b).

When exceptions occur, a block of statements is invoked implicitly—that is, without an explicit call. This is the distinguishing feature of exceptions. The same effect could be obtained through explicit calls by including conditional calls to the handling routines. For example, before each array access, a statement could be placed that is of the form

\[
\text{if index > index\_max or index < index\_min then}
\]
\[
\text{out\_of\_range\_handler;}
\]

Such frequently recurring statements would detract from the understandability of the program, however. Therefore, implicit invocation is desirable.

Not all imperative languages contain a facility for exception handling, but most recent ones, including Java, C++, and Ada do. Our discussion in this section will focus on the general approach rather than on a single language's implementation. In Section 7.10, the Java approach will be described.

### 7.9.1 Raising exceptions

Due to the implicit nature of the invocation of exception handlers, no special syntactic features are required for their invocation. The implicit invocation of an exception is commonly referred to as raising or throwing the exception. Most languages do, however, provide a facility for the explicit invocation of exceptions as well. This facility is in the general form of a RAISE statement and consists of a keyword such as raise, followed by the name of the exception. This convention permits the program to explicitly raise a built-in exception in order to invoke the exception handler at that point even without the associated condition occurring.

In addition to the built-in exceptions provided by a language, the language may also permit user-defined exceptions. Such exceptions can be declared like any other data object and will exist within the scope of their declaration. User-defined exceptions must be raised explicitly, because they have no associated synchronous condition. These exceptions thus behave much like procedures and may even be permitted to accept parameters. The major difference between user-defined exceptions and procedures lies in the flow of control on termination of the invoked unit. Procedures will always return to the point immediately after the point of invocation, whereas an explicitly invoked exception handler may proceed differently. The possibilities are discussed shortly.

A language might permit the enabling and disabling of exceptions. Whether this option is available or not is a philosophical decision the language designers must make. Suppression of an exception through disabling might be of value when testing for the exception is expensive in time and space. Also, the value of the resulting reliability is not worth that cost to the programmer. Index-out-of-range testing for arrays is an example of an exception whose cost might exceed its
value in some circumstances. In these cases, some languages permit specified exceptions to be
disabled in the scope of a program unit. This suppression occurs at compile time and prevents the
generation of the code needed to implement the exception.

7.9.2 Handling exceptions

Exception handlers are blocks of statements that are bound to an exception. Figure 7.18 illustrates the bindings of exceptions. For built-in exceptions, the name binding is a permanent part of the language. A user-defined exception is bound to its name at the point of declaration, and that binding holds within the scope of that declaration.

**Figure 7.18 Exception Object**

![Diagram of Exception Object]

The binding of the exception to its exception handler can follow either of two models. First, the binding can be patterned after data/object-name bindings within program units. In this case, the exception-handling block is bound to the exception in the scope of a program unit through a declaration attached to that unit. A redefinition of the exception handler in a contained unit will temporarily interrupt that binding, giving an exception version of a hole-in-scope.

The second model is similar to the data/Object-value binding, where statements executed within the block can modify the exception/exception handler binding.

To illustrate these two models, consider the pseudocode illustrations in Figures 7.19 and 7.20. In Figure 7.19a, exceptions are declared, and they are separately assigned handlers in the declaration section of the program unit. The binding of a handler to an exception holds throughout the scope of the unit. The handler for exception \( E_1 \) is redeclared in procedure \( B \), introducing a hole-in-scope for the \( H_1 \) handler within \( B \). Here we see that the name binding \( E_1 \) holds throughout \( A \) although the handler binding changes within \( B \). This model is implemented in Ada. A variation of this model binds the exception to the handler at the bottom of the program unit as illustrated by the pseudocode example in Figure 7.19b. This is the model used by Java.

**Figure 7.19a Illustration of Model 1 Exception Binding**

```plaintext
procedure A
  E1, E2 : exception;
  handler E1 is
```
Figure 7.19b Illustration of Model 1 Exception Binding

procedure A;
   E1, E2 : exception;
procedure B;
   E3 : exception;
begin --B
   <B Block-of-Statements>
   -- E1 handler is H4
   -- E2 handler is H2
   -- E3 handler is H3
end; --B
begin --A
   -- E1 handler is H1
   -- E2 handler is H2
   B;
end; --A

Handlers
   E1: <H4 Block-of-statements>
   E2: <H2 Block-of-statements>
   E3: <H3 Block-of-statements>
end; --B
begin --A
   <A Block-of-statements>
   B;
   -- E1 handler is H1
The second model is illustrated by Figure 7.20. In this pseudocode example, the handler is not bound to the exception by a declaration but rather by an executable statement. The statement we create to do this, the `SET HANDLER` statement, acts much like an assignment statement by binding an exception handler to an exception at run time. A difficulty with this second model is illustrated by the example. The environment of the call affects the binding of the handler. In Figure 7.20, exceptions E1 and E2 have different initial bindings for the two invocations of B. This model is implemented in PL/I.

**Figure 7.20 Illustration of Model 2 Exception Binding**

```plaintext
procedure A;
    E1, E2 : exception;
procedure B;
    E3 : exception;
begin --B
    -- E1 handler is H1 (first call) or H4 (second call)
    -- E2 handler is H2 (first call) or H5 (second call)
    set handler E3 to
        <H3 Block-of-statements>
    -- E3 handler is H3
    ...
    set handler E1 to
        <H4 Block-of-statements>
    -- E1 handler is H4
    ...
end; --B
begin --A
    set handler E1 to
        <H1 Block-of-statements>
    -- E1 handler is H1
    set handler E2 to
        <H2 Block-of-statements>
    -- E2 handler is H2
    ...
B;
    -- E1 handler is H4
```
... set handler E2 to
    <H5 Block-os-statements>
    -- E2 handler is H5
    B;
end;   --A

The flow of control upon completion of the exception handler is a rather complex issue. There are four alternative approaches here.

1. Terminate the program unit that invoked the exception, returning to that unit's invoking unit.
2. Terminate the program unit that invoked the exception as in (1) and raise the same exception in the unit's invoking unit.
3. Resume execution of the program unit that invoked the exception by retrying the statement or expression where the exception occurred.
4. Resume execution of the program unit that invoked the exception at the point immediately following the statement where the exception occurred.

Languages may determine which of the above actions are used through one of the following policies:

1. Use the same default action on completion of all exceptions.
2. Each exception type has a given default completion action associated with it, but different built-in exceptions may have different default actions.
3. The language includes constructs for specifying the completion action within the exception handler.

A related issue is the raising of an exception for which no handler has been defined. In this case, there may either be a default handler associated with every exception whose handler has not been explicitly defined or the exception may be propagated.

**Exception propagation** occurs as follows. If an exception is raised when no handler is specified, the exception results in the termination of the program unit that raised the exception and raises the same exception in the unit that invoked the current unit. The same exception--which is now raised in the invoking unit--may have a defined handler in that unit, in which case that handler is executed. If there is no handler there, that unit is terminated, and the same exception is raised in its invoking unit. Thus, an unhandled exception continues to propagate up levels of the dynamic invocation tree until it reaches a unit where it is handled. If it reaches the root of the tree and still is not handled, a default handler will be executed resulting in the root unit being terminated.

Another way of dealing with unhandled exceptions is to permit the programmer to specify a default handler that is invoked when any exception is raised that has no defined handler in the present context.
7.9.3 Implementation

The implementation of an exception depends on its type and the way it is specified in the language. The implementation of exception raising can be performed by hardware interrupts in some cases, operating system traps in others, and compiler-generated code inserted into the program in others. Which technique is used for a given exception depends upon the hardware and software capabilities of the system on which the language is implemented.

As for the activation record, a list of built-in and user-defined exceptions is included as a part of the activation record of each program unit. Each exception entry in the table includes a pointer to the associated handler if one is defined in the present program unit. When an exception is raised, the table is searched for the appropriate exception name, and if it is found, control is transferred to the code for the associated handler. If no handler is specified, either a default handler is executed or the exception is propagated. If the exception is not found in the list of defined exceptions, a search of the nonlocal environment through the static pointer chain can proceed just as with other data objects, if that is the action prescribed by the language.

Exception propagation is implemented by terminating the presently active program unit, popping its activation record off the run-time stack, and raising the same exception in its invoking unit, which is represented by the present top-of-stack activation record. This propagation will continue down the dynamic chain until an activation record is found that handles the exception.

The implementation of the exception handler depends on the action taken upon completion of the handler. Handlers that resume execution of the invoking unit can be implemented just like procedures with activation records of their own. Handlers that terminate the invoking unit are typically implemented like a goto within the same unit, with a return from the active unit executed at the completion of the handler causing the present activation record to be popped off of the run-time stack.

7.10 Exceptions in Java

Java exceptions are instances of the Java class Throwable or one of its subclasses. There is a collection of exception classes included in the standard Java API. User-defined exceptions may be created by creating new classes in the Java exception class hierarchy.

Java exceptions are handled within a try statement. The general form is

```java
try {
    <body_statement_block>
} catch (<exception_class> <argument_identifier>) {<exception_handler_statement_block> }
[finally {finally_statement_block} ]
```

When an exception is thrown during the execution of the body_statement_block, the class of that exception is checked against the exception_class of each catch clause. The first exception_class that is the class of the thrown exception or a superclass of that class trig-
gers the execution of the corresponding exception_handler_statement_block, followed by the execution of the finally_statement_block, if present. If no such matching catch clause is found, the finally_statement_block is executed, if present, the enclosing execution unit is terminated, and the same exception is thrown in the invoking unit at the point of invocation. This requires that the enclosing unit’s signature include a throws clause that encompasses the thrown exception class. If the body_statement_block is terminated without the exception being thrown, the finally_statement_block is executed, if present, and the execution continues at the point immediately following the try statement.

Java recognizes two kinds of exceptions: checked and unchecked. Checked exceptions are those that must be handled in every block of code where the exception can possibly occur. For example, every file input statement must be contained in a block that catches the end-of-file exception. This is enforced by the compiler.

Unchecked exceptions are those that do not require every possible occurrence to be enclosed in a try statement with a corresponding catch. For example, the Index_Out_Of_Bounds exception is not required to be caught everywhere there is an index reference.

Some of the more common unchecked exceptions in the Java Standard API are ArithmeticException, ClassCastException, NumberFormatException, IllegalArgumentException, IndexOutOfBoundsExceptions, and NullPointerException.

Some common checked exception classes are ClassNotFoundException, EOFException, and FileNotFoundException.

Java includes the throw statement for explicitly throwing an exception. The form of this statement is

```java
throw <exception_name>;
```

where exception_name is an instance of some exception class.

**Discuss:** List some exceptions for which each of the four policies for flow of control after exception handling would be appropriate.

- a. Terminate program unit and return to invoking unit.
- b. Terminate program unit and raise same exception in invoking unit.
- c. Resume execution by retrying statement where exception was raised.
- d. Resume execution immediately after statement where exception was raised.

**Discuss:** In Ada, exception suppression occurs at compile time rather than run time. What advantage would run-time suppression provide? What difficulties would it present in implementation?

**Reinforce:** Describe how a general while iteration could be simulated by Java exceptions plus a Java non-terminating iteration as they are implemented in Java.

**Laboratory:** 1. Write a program which shows whether parameters of a VRP can be
reference parameters in the language assigned to you.

Laboratory: For each built-in exceptions in your language, generate a situation that will raise it.

Laboratory: Write a binary search procedure using exceptions instead of controlled iteration statements.

Laboratory: Determine what happens in your language if an exception is raised within an exception handler.

Terms - Chapter 7

abstraction
activation record
run-time stack
local environment
nonlocal environment
static scope
dynamic scope
side effect
parameters
positional parameter association
named parameter association
default parameter association
IN parameter
OUT parameter
IN OUT parameter
actual parameter
formal parameter
positional parameter association
named parameter association
aliasing
value returning procedure
overloading
coroutine
exception
synchronous
asynchronous
exception handler
raising exceptions
enabling
disabling
exception propagation