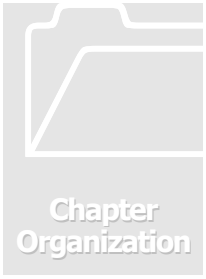
Chapter 5

5. Beyond The Basics

Read this chapter to learn about advanced language concepts.

This chapter introduces more advanced language concepts as a starting point for your exploration of system and application programming with CM3-IDE.

This chapter treats the language concepts pedagogically; each concept is presented informally and is illustrated through a complete example program. The description of concepts and features in this chapter is incomplete; for complete and precise definitions of language features, see the *Language Reference*.



This chapter is divided into six parts. Each part describes a distinct feature of the language, and demonstrates the feature in a complete program. You can find the sources for the programs in this chapter in the Examples section of your CM3-IDE environment.

Exceptions: Error Handling in CM3-IDE on page 74 illustrates the use of exceptions for building robust programs.

Object Types: Object Oriented Programming on page 81 showcases basic object-oriented programming in a simple program.

Threads: Managing Concurrent Activities on page 86 describes how to use threads to manage concurrent activities.

Opaque Types: Information Hiding and Encapsulation on page 88 demonstrates the use of opaque types to hide the implementation of a type from its clients. The section continues with partially opaque types, which can be used to reveal partial information about objects to select clients.

Generics: Reusable Data Structures and Algorithms on page 97 outlines the use of parametrized interfaces for creating polymorphic data structures and algorithms.

Unsafe Constructs: System Programming in CM3-IDE on page 102 introduces you to the unsafe subset of the language.

5.1 Exceptions: Error Handling in CM3-IDE

Error handling in CM3-IDE is done with language-level exceptions. Exceptions help separate error handling logic from the main code.

If you want your program to remain vigilant at all times for errors where exceptions are not available, you need to explicitly check for failure at every function call in your program. Due to the usual pressures of software development, returning or checking error codes for failure is seldom done thoroughly. Careful programmers who check for every error condition often design elaborate arrangements of if-then statements, or non-standard "`setjmp`" calls. Most of these methods make programming much more difficult for the careful programmer, hence encouraging sloppy treatment of errors.

Error-checking with exceptions enables the programmer to easily, and reliably handle all error conditions in their programs.

5.1.1 How Exceptions Work

In CM3-IDE, when a program encounters a situation deemed abnormal by the programmer, the runtime generates an exception, and begins to look for a handler for the exception to handle the abnormal condition.

If an exception is not handled in the current procedure, the calling procedure is searched. If no handler for that particular exception is found there, the runtime will continue following the chain of called procedures until it finds a procedure that handles that particular exception. If no handler is found, the computation terminates, for example, by entering the debugger.

Hence, it is possible to handle errors anywhere in a chain of procedure calls, without having to continually check error codes.

For a precise definition of the semantics of exceptions, see the *Language Reference*. Here we describe the syntax for raising and handling exceptions, and illustrate the use of exceptions in a simple program.

5.1.2 Declaring Exceptions

All exceptions must be declared at the top-level of an interface or a module. An exception may include a parameter.

```
EXCEPTION exception-name [ "(" exception-parameter ")" ];
```

In the following example, `DriveNotReady` is a parameter-less exception and `ReadError` is an exception that takes a TEXT parameter:

```
INTERFACE CDROM;

EXCEPTION DriveNotReady;
EXCEPTION ReadError(TEXT);

PROCEDURE Read(sector: INTEGER):TEXT
   RAISES {DriveNotReady, ReadError};
END CDROM.
```

5.1.3 Triggering Exceptions: RAISE Statement

You can trigger the handling of an exception through the use of the RAISE statement. The statement:

```
RAISE exception [ "(" exception-parameter ")" ];
```

raises an exception, passing control of the program to the innermost exception handler for that exception. Use the RAISE statement to raise an exception. If the exception is defined with a parameter, one must be supplied when the exception is raised.

5.1.4 Handling Exceptions: TRY-EXCEPT Statement

```
TRY
   guarded-statements
EXCEPT
 "|" exception-name { "," exception-name … } "=>" action-statement
 "|" exception-name "(" parameter-name ")" "=>" action-statement
[ ELSE statements ]
END
```

The TRY-EXCEPT statement guards statements between TRY and EXCEPT with the exception handlers between EXCEPT and END.

An exception raised by a guarded-statement is handled by the action-statement which has a matching handler for the exception, or by the ELSE clause, if present. If an exception is caught, execution continues with the statement following the END, otherwise the exception is passed on to the enclosing scope.

Example:

```
EXCEPTION Failure(Severity);
TYPE Severity = {Low, Medium, High};
…
TRY
…Code that may raise Failure, IO.Error, or Lex.Error …
EXCEPT
| IO.Error   => IO.Put("An I/O error occurred.")
| Lex.Error  => IO.Put("Unable to convert datatype.")
| Failure(x) => IF x = Severity.Low
                  THEN IO.Put("Not bad")
                  ELSE IO.Put("Bail out")
               END
END;
```

Important Note. The language defines RETURN and EXIT in terms of exceptions, hence the ELSE clause of a TRY-EXCEPT statement may catch a RETURN or an EXIT. You should refrain from using the ELSE clause of a TRY-EXCEPT statement whenever possible.

5.1.5 Cleaning up: TRY-FINALLY Statement

The TRY-FINALLY statement is typically used for clean-up activities.

```
TRY
   guarded-statements
FINALLY
   cleanup-statements
END
```

TRY-FINALLY guarantees that cleanup-statements are called no matter what happens in the guarded-statements. If one of guarded-statements raises an exception, the same exception is re-raised by TRY-FINALLY after the cleanup-statements are executed.

TRY-FINALLY is useful for clean-up activities, like closing file handles, even when the I/O operations may fail:

```
rd := IO.OpenRead("myfile");
TRY
   WHILE NOT EOF(rd) DO
      IO.PutLine (IO.GetLine(rd));
   END;
FINALLY
   Rd.Close(rd);
END;
```

Important Note. As RETURN and EXIT semantics are defined in terms of exceptions, TRY-FINALLY will not only act upon an ordinary exception, but it also acts upon a RETURN or an EXIT. This is often handy for adding wrappers to a block of code to print debugging information no matter how the block of code behaves.

5.1.6 Trapping All Exits from a Block of Code

For example, starting with a procedure with multiple return paths:

```
PROCEDURE SomeProc(): BOOLEAN RAISES {Invalid} =
BEGIN
   FOR i := 1 TO 10 DO
   CASE option[i] OF
    | 'a'..'z' => RETURN TRUE;
    | '1'..'9' => EXIT;
    ELSE RAISE Invalid;
   END;
   RETURN FALSE;
END SomeProc;
```

Suppose you would like to print a message every time a procedure exits, no matter how it exits: It is easy to add a `TRY-FINALLY` statement to catch every exit from the procedure. Simply add a `TRY-FINALLY` around the procedure's body; the `FINALLY` clause will be called no matter how the program exits `SomeProc`'s scope.

5.1.7 An Example of Exception Handling

The next two programs illustrate how to use exceptions to make your programs more robust against failures. The first one is a simple file copy program that does not deal with exceptions. The second incarnation catches exceptions, and hence is more robust.

5.1.8 Programming without Exceptions

Here we review the implementation of the `Copy` program. This program may crash at run-time due to uncaught exceptions.

Indeed, if you compile the sources for this version of the copy program, you will notice a number of warnings regarding possible exception failures at run-time. It is easy to cause a run-time crash: simply attempt to copy a non-existent file. The compiler warnings notify you about possible run-time failures at *compile-time*. If you fix all the exception-related warnings in all the sources of a program, your program will never crash from an unhandled exception.

Let's start with the main module, named `Copy`. The `Copy` program is simple:

- Make sure that the user has specified arguments correctly.
- If parameters are wrong, return an error code and exit.
- Otherwise, pass them along to `FakeOS.Copy`.

```
MODULE Copy EXPORTS Main;
   IMPORT FakeOS, Params, Process, IO;
BEGIN
   IF Params.Count # 3 THEN
      IO.Put ("Syntax: copy <source> <destination>\n");
      Process.Exit (2);
      END;
   WITH source = Params.Get(1) DO
      WITH destination = Params.Get(2) DO
         FakeOS.Copy (source, destination);
      END;
   END;
END Copy.
```

As a user-defined interface, `FakeOS` provides access to the `FakeOS` module. Copy a file named source to a file named destination.

```
INTERFACE FakeOS;
   PROCEDURE Copy(source, destination: TEXT);
END FakeOS.
```

The `FakeOS` module supplies the body of the `Copy` procedure. The `Copy` procedure creates new reader and writer streams from the input and output files, reads the contents from the input, and writes it to the output.

The procedures `FileRd.Open` and `FileWr.Open` are used for reading and writing files, `Rd.GetText` and `Wr.PutText` for input and output, and `Wr.Close` and `Rd.Close` to close the I/O streams.

`FakeOS.Copy` does the following:

- Open a reader and a writer to the source and destination
- Read the contents of the reader
- Write the contents into the writer
- Flush and close the reader and the writer

```
MODULE FakeOS;
   IMPORT Rd, Wr, FileRd, FileWr;
PROCEDURE Copy(src, dest: TEXT) =
   VAR rd: Rd.T; wr: Wr.T;
BEGIN
   rd := FileRd.Open (src);
   wr := FileWr.Open (dest);
   WITH contents = Rd.GetText (rd, LAST(INTEGER)) DO
      Wr.PutText (wr, contents);
   END;
   Rd.Close (rd);
   Wr.Close(wr);
END Copy;

BEGIN
END FakeOS.
```

Note that `FakeOS.Copy` does not handle any exceptions raised by `FileRd.Open`, `FileWr.Open`, `Rd.GetText` or `Wr.PutText`. You may review the definition of these procedures in the standard libraries to find out about the exceptions they may raise. You can easily create a situation where an uncaught exception, causes a run-time crash.

5.1.9 Making Programs Robust with Exceptions

The previous version of `FakeOS.Copy` has a serious problem: it crashes when any I/O exception (e.g., disk full, no permission to write) is raised. The new version of the `FakeOS` interface and implementation illustrates how to use exceptions to build robust programs. The main module `Copy` still calls the `FakeOS` interface. The required changes are:

1. Add an `Error` exception with a `TEXT` parameter to the `FakeOS` interface.

2. Change `FakeOS.Copy` to raise `Error` when there is a problem, and include a text string describing the problem.

3. Modify the `Copy` module to handle this exception by printing an error message for the user.

```
INTERFACE FakeOS;
EXCEPTION Error (TEXT);
PROCEDURE Copy(source, destination: TEXT) RAISES {Error};
END FakeOS.
```

Here is the implementation of the new FakeOS:

```
MODULE FakeOS;
IMPORT Rd, Wr, FileRd, FileWr;
IMPORT Thread, OSError;
```

FakeOS works similarly to the last version, but this time it catches exceptions using the TRY-EXCEPT clause. For each exception that is raised by the called procedures, we propagate an Error exception to the caller of FakeOS.Copy.

```
PROCEDURE Copy(src, dest: TEXT) RAISES {Error} =
VAR
   rd: Rd.T;
   wr: Wr.T;
<* FATAL Thread.Alerted *>
BEGIN
   TRY
      rd := FileRd.Open (src);
      wr := FileWr.Open (dest);
      WITH contents = Rd.GetText (rd, LAST(INTEGER)) DO
         Wr.PutText (wr, contents);
      END;
      Rd.Close (rd);
      Wr.Close(wr);
   EXCEPT
    | Rd.Failure =>
         RAISE Error ("reading from " & src & " failed");
    | Wr.Failure =>
         RAISE Error ("writing to " & dest & " failed");
    | OSError.E =>
         RAISE Error ("a system problem occured");
   END
END Copy;

BEGIN
END FakeOS.
```

Note that the exception Thread.Alerted is marked as fatal, because we didn't want to handle it. The FATAL pragma lets the programmer tell the compiler that a particular exception is intentionally not being handled. You should employ the FATAL pragma carefully; excessive use of the FATAL pragma results in crash-prone code. If FakeOS.Copy was to deal with multiple threads, it should deal with Thread.Alerted properly.

What happens to the Close statements if there is an exception raised while the files are open? Yes, that's a problem—if an exception occurs while copying, the files may be left open—and you can use a TRY-FINALLY statement to deal with it. Review the language reference or tutorial to learn about TRY-FINALLY.

While these kinds of issues are usually not important in short-lived programs, they are very important for long-lived, multi-threaded applications (for example, a network object server) where resource management is critical.

This program is now robust against various system exceptions raised by calls in `FakeOS` or `Copy` modules. If the program hadn't been handling a particular exception, you would have seen a warning at compile-time. This same program, without source modifications, will work without silent or unexpected errors due to system exceptions on all supported operating systems.

5.2 Object Types: Object-oriented Programming

This section assumes that you are familiar with the concepts of object-oriented programming.

An object associates some *state* with some *behavior*. A Modula-3 object is a record—its state—paired with a method suite—its behavior.

```
TYPE
   AnObjectType "=" [ parent-object-type ] OBJECT
      object-fields
   [ METHODS methods ]
   [ OVERRIDES overrides ]
   END
```

An object is a record paired with a *method suite*, a collection of procedures that operate on the object. The fields of an object are specified just like those of records. Methods look like fields that hold procedure values, with the following syntax:

```
methods = { method ";" … }
method = identifier signature [ ":=" procedure-name ]
```

A *method signature* is similar to a procedure signature (See the section on procedure declarations in the *Language Reference*). If a method declaration includes "`:=`", the associated procedure must accept objects of this type as its first parameter; the rest of the parameters much match the signature. The first parameter is often referred to as the *self* parameter.

Overrides specify new implementations for methods declared by an ancestor object-type: (An ancestor is either the parent of this type, or its parent's parent, or ...)

```
overrides = { override ";" … }
override = method-name ":=" procedure-name
```

Object types form a single-inheritance hierarchy. The type `ROOT` is the supertype of all object types. Objects are traced references by default, hence they are garbage-collected by default.

Example. Let's create an object type `Polygon` which contains an open array of coordinates. We also define an initialization method `init`, and a verification method `verify`, for all objects of this type. Subtypes of `Polygon` may override the `init` method, and must override the `verify` method.

```
TYPE
   Polygon = OBJECT
      coords: REF ARRAY OF Point.T;
   METHODS
      init( READONLY p: ARRAY OF Point.T): Polygon := Init;
      verify() := NIL; (* To be overridden by subclasses. *)
   END;

PROCEDURE Init (self: Polygon;
                READONLY p: ARRAY OF Point.T) =
BEGIN
   self.coords := NEW(NUMBER(p));
   self.coords^ := p;
   self.verify(); (* Verify that initialization was proper. *)
   RETURN self;
END;
```

The subtype `Drawable` adds the `draw` method and assigns the `Draw` procedure as the default implementation for the `draw` method.

```
TYPE
   Drawable = Polygon OBJECT METHODS
      draw() := Draw;
   END;

PROCEDURE Draw (self: Drawable) =
BEGIN
   WITH p = self.coords^ DO
      FOR i = FIRST(p) TO LAST(p) DO
         DrawLine(p[i], p[(i+1) MOD NUMBER (p)])
      END;
   END;
END Draw.
```

Type `Rectangle` is a concrete implementation of an object. It will override the `verify` method to make sure there are four sides to this polygon and that the sides have the right properties.

```
TYPE
   Rectangle = Drawable OBJECT METHODS
   OVERRIDES
      verify := Verify;
   END;

PROCEDURE Verify (self: Rectangle) =
BEGIN
   WITH p = self.coords^, dist = Point.DistSquare DO
      <* ASSERT NUMBER(p) = 4 *>
      <* ASSERT dist (p[0],p[2]) = dist (p[1],p[3]) *>
   END
END Verify;
```

Assuming point is a `ARRAY [1..4] OF Point.T`, to draw a new `Rectangle` object, you must do:

```
VAR
   rect: Rectangle;
BEGIN
   rect := NEW(Rectangle);
   rect.init(points);
   rect.draw();
END
```

Or the shorthand:

```
VAR
   rect := NEW(Rectangle).init(points);
BEGIN
   rect.draw();
END
```

5.2.1 Programming with Objects: A Complete Example

The complete program `Objects` is another example of the use of objects.

```
MODULE Objects EXPORTS Main;
IMPORT IO;
```

The type `Person` declares a new object type with fields `firstname`, `lastname`, and `gender`. `Person` also defines a method `fullname()` which is implemented by procedure `FullName`.

```
TYPE
   Person = OBJECT
      firstname, lastname: TEXT;
      gender: Gender;
   METHODS
      fullname(): TEXT := FullName;
   END;
   Gender = {Female, Male};

PROCEDURE FullName (self: Person): TEXT =
   CONST Title = ARRAY Gender OF TEXT {“Ms.”, “Mr.”};
BEGIN
   RETURN Title[self.gender] & “ ” & self.firstname &
      “ ” & self.lastname;
END FullName;
```

Any code that can see the declaration of the object type `Person` can create new instances of that type. So, anywhere in this module, you can create a new instance of the type `Person`. (You can use interfaces to control the visibility of object types. See **Opaque Types: Information Hiding and Encapsulation** on page 88 for more information.)

Here a new object is assigned to the variable `john`.

```
VAR
   john := NEW(Person,
               gender := Gender.Male,
               firstname := "John",
               lastname := "Smith");
```

Here is a procedure, `Describe`, which takes a `Person` object and a text description and prints a line to the standard output using the `fullname()` method.

```
PROCEDURE Describe (person: Person; description: TEXT) =
BEGIN
   IO.Put (person.fullname() & " is " & description & ".\n");
END Describe;
```

`Describe` calls the `fullname` method of its first parameter. Of course, different `Person` objects can have different implementations of the `fullname` method, so, ultimately you can pass different subtypes of `Person` into this procedure. Here we create one such subtype, named `Employee` which has some additional fields. Note that it shares the same implementation as `Person` for the `fullname()` method.

```
TYPE
   Employee = Person OBJECT
      company: TEXT;
   END;
```

Next, we create a new instance of this type, named `jane`. You can list the fields of an object in any order when you initialize it.

```
VAR
   jane := NEW(Employee,
               firstname := "Jane",
               lastname := "Doe",
               company := "ACME Ltd",
               gender := Gender.Female);
```

You can create new subtypes that override existing methods. In the next subtype, the `fullname()` method of `Doctor` object, implemented via `FullDoctorName`, skips the first name and uses a professional title for referring to a `Doctor` instance. Note that `PrintDoctorName`'s self argument is of type `Doctor`.

```
TYPE
   Doctor = Person OBJECT
      title: TEXT := NIL;
   OVERRIDES
      fullname := FullDoctorName;
   END;
```

```
PROCEDURE FullDoctorName(self: Doctor): TEXT =
   VAR
      result: TEXT := "Dr. " & self.lastname;
BEGIN
   IF self.title # NIL THEN
      result := result & ", " & self.title & ", ";
   END;
   RETURN result;
END PrintDoctorName;
```

Let's create a couple of instances of `Doctor`.

```
VAR
   dr_who := NEW(Doctor,
                 lastname := "Who",
                 title := "Time Lord");

   dr_quinn := NEW(Doctor,
                   lastname := "Quinn",
                   title := "Medicine Woman");
```

There is also a shorthand for creating one-of-a-kind objects types as part of a `NEW` call. That's how `joe` gets created.

```
VAR
   joe := NEW(Person, firstname := "Joe",
                      lastname := "Schmo",
                      fullname := AnAverage);

PROCEDURE AnAverage(self: Person): TEXT =
BEGIN
   RETURN "An average " & self.firstname & " " &
      self.lastname;
END AnAverage;
```

Finally, make a few calls to `Describe` just to show that it works.

```
BEGIN
   Describe (john, "a nice person");
   Describe (jane, "an employee of " & jane.company);
   Describe (dr_who, "a bit weird");
   Describe (dr_quinn, "not for real");
   Describe (joe, "probably good enough for working" &
      "on this project");
END Objects.
```

Feel free to create your own subtypes of `Person`!

For more information on controlling visibility of object declarations, see **Opaque Types: Information Hiding and Encapsulation** on page 88.

5.3 Threads: Managing Concurrent Activities

A *Thread* is the fundamental concurrency abstraction in Modula-3.

Using threads, you can create concurrent activities within a single activation of a program.

Threads can be very useful for structuring real-world programs, because they often need to deal with multiple activities. For example, a program that needs to interact with the user and manage a long-running query to a database can use a thread for each task, so that one task can progress without waiting for the other tasks.

Threads enforce a separation of concerns of concurrent activities in your program, which helps you manage each task better.

Using the required `Thread` interface, you can create and manage threads in an operating-system-independent manner. The `Thread` interface provides operations for communication and synchronization between threads. Standard libraries distributed with your system are thread-friendly. Indeed, some libraries, such as the user interface toolkit `Trestle`, use threads in order to perform their background activities.

For a good introduction to threads, read Andrew Birrell's article, *Introduction to Programming with Threads*, included on-line as part of your CM3-IDE distribution. The sample program `ThreadExample` briefly outlines how you can program with multiple threads.

`ThreadExample` is a simple multi-threaded program:

```
MODULE ThreadExample EXPORTS Main;
IMPORT Thread, Fmt, IO;
```

Driven by commands from the standard input. For each user request, the program spawns a new thread which waits for a specified elapsed time.

ThreadClosure. The following fragment creates a new *closure* object, which embodies the state of a thread. In this case, the `Thread.Closure` subtype `TimeClosure` contains the state required for a timer thread: a length of time that a thread must pause.

By convention, thread closures override the `apply` method to designate the work to be done. `TimerClosure`'s implementation is in procedure `TimerApply`.

```
TYPE
   TimerClosure = Thread.Closure OBJECT
      time: LONGREAL;
   OVERRIDES
      apply := TimerApply;
   END;
```

`TimerApply` performs the work of the timer threads:

- print out a message that it has started
- wait for `time` seconds
- print out a message that it has finished.

The local variable `count` allows the user to match the start and finish messages.

```
PROCEDURE TimerApply (cl: TimerClosure): REFANY =
VAR
   count := Counter();
BEGIN
   Print("\nStarting timer " & Fmt.Int(count) &
      " for " & Fmt.LongReal (cl.time) & " seconds.");
   Thread.Pause (cl.time);
   Print ("\nFinished timer " & Fmt.Int(count) &
      " after " & Fmt.LongReal (cl.time) & " seconds.\n");
   RETURN NIL;
END TimerApply;
```

Thread Synchronization. The variable `timer_count` keeps track of the count for the threads created; `timer_count_mu`, a *mutex* (or a *lock*) protects the critical sections, where multiple threads may be contending for `timer-count`.

```
VAR
   timer_count: CARDINAL := 0;
   timer_count_mu := NEW(MUTEX);
```

`Counter` returns a new counter. `Counter`'s critical section (the place where multiple threads may be racing each other) is protected by a `LOCK` statement. Note that `mutex` is automatically unlocked upon exit from the scope of the `LOCK` statement, so `RETURN timer_count` effectively unlocks `timer_count` on its way out of the procedure.

```
PROCEDURE Counter(): CARDINAL =
BEGIN
   LOCK timer_count_mu DO
      INC (timer_count);
      RETURN timer_count;
   END;
END Counter;
```

Forking Threads. The main program waits for user input and forks threads for new timers when the user asks for one.

The main loop reads input from the user to determine how long the next forked thread should pause. A `closure` is created dynamically to be passed into `Thread.Fork`, which will fork a new thread and run `closure.apply()`.

```
VAR
   input: CARDINAL;
   closure: TimerClosure;
BEGIN
   LOOP
      IO.Put(Prompt);
      input := IO.GetInt();
      closure :=
         NEW(TimerClosure, time := FLOAT(input, LONGREAL));
      EVAL Thread.Fork (closure);
   END;
END ThreadExample.
```

There was no need to wait for the forked thread in this example. To wait for a forked thread to complete, you call `Thread.Join(th)` which returns the value returned by the thread's `apply` method.

```
th := Thread.Fork (cl);
  … do other activities …
result := Thread.Join (th)
```

Other calls in the `Thread` interface, such as `Signal`, `Wait`, and `Broadcast` provide for more intricate synchronization patterns. For a thorough introduction, see Andrew Birrell's article, *Introduction to Programming with Threads*, available on-line in the Technical Notes section of your CM3-IDE distribution.

5.4 Opaque Types: Information Hiding And Encapsulation

Encapsulating implementation details is a key technique in managing the growth of large programs. Often, you may want to reveal references to data structures in a module, while hiding the structure and implementation of a datatype. Opaque types are a mechanism for enforcing such a separation. An *opaque type* is a name that denotes an unknown subtype of a known reference type. For example, all you know about an opaque subtype of `REFANY` is that it is a traced reference. (`REFANY` is the root of the traced heap.) The actual type denoted by an opaque type name is called its *concrete type*.

Different scopes can reveal different information about an opaque type. For example, what is known in one scope only to be a subtype of `REFANY` could be known in another scope to be a subtype of `ROOT`.

An opaque type declaration has the form:

```
TYPE T <: U
```

where `T` is an identifier and `U` an expression denoting a reference type. It introduces the name `T` as an opaque type and reveals that `U` is a supertype of `T`. The concrete type of `T` must be revealed elsewhere in the program.

5.4.1 Fully Opaque Types

The simplest way to hide information is to divide your program into two groups: portions of your code where everything about the structure of a type is revealed, and portions where nothing is revealed. This dichotomy is the essence of *fully opaque types*. A fully opaque type is a subtype of `REFANY`, or `ADDRESS`, corresponding to a traced or untraced reference to an unknown type.

By combining fully opaque types with interfaces, you can create abstract datatypes with full encapsulation: the interface pairs the name of the type, with a set of procedures that operate on that type.

In the next example, the `Person` interface exports an opaque type `Person.T`, and the operations `New` and `Describe`.

```
INTERFACE Person; IMPORT Wr;
```

Declare `Person.T` as an opaque type.

```
TYPE
   T <: REFANY;
```

The `Person` interface defines an opaque type `T` (often called an *abstract data type*) with two operations:

- `New` for creating new instances
- `Describe` for printing textual descriptions.

Since none of the clients can "see through" this opaque interface, it should describe precisely *what* the implementation does without revealing *how* the implementation works.

The statement `U <: V` means that `U` is a subtype of `V`. When you see such a declaration, you can assume that an instance of `U` supports at least as many operations as an instance of `V`. In this case, since `V` is `REFANY`, all you can assume about `Person.T` is that it is a traced reference. It's traced, so you don't have to worry about managing its memory. It's a reference so you can store it, or compare it with another reference of the same type for equality.

```
Gender = {Female, Male};
```

`Person.Gender` is an enumeration type with elements `Female` and `Male`.

```
PROCEDURE New (firstname, lastname: TEXT;
               gender: Gender): T;
```

Create a new `Person.T` given a first name, a last name, and a gender.

```
PROCEDURE Describe(person: T; desc: TEXT; wr: Wr.T := NIL);
```

Write a textual description, `desc`, of a `Person.T` to the writer stream `wr`. If `wr` is not specified, write the description to the standard output.

```
END Person.
```

Note that nothing about the implementation of `Person.T` is visible to clients of `Person`. Indeed, the compiler will not know any more information while compiling clients of this interface; hence, if you change the implementation for this module, you don't have to recompile its clients.

```
MODULE Person;
IMPORT IO, Wr;
```

The `Person` module implements the opaque type `Person.T`. To do so, it reveals the representation of `Person.T` completely. Within the module, we can use the items declared in the interface without qualification. Hence, `T` in this module refers to `Person.T` and `Gender` refers to `Person.Gender`.

Since the only information specified in the `Person` interface is that `Person.T` is a reference—or more precisely, `Person.T <: REFANY`—the implementation can specify the full structure of the object. Indeed, the full revelation of `Person.T` is similar to an ordinary object type declaration.

```
REVEAL
   T = BRANDED OBJECT
      firstname, lastname: TEXT;
      gender: Gender;
   METHODS
      fullname(): TEXT := FullName;
   END;
```

The `BRANDED` keyword is required in all full revelations and ensures that instances of `Person.T` are distinct from all other types with the same structure. Essentially, the `BRANDED` keyword overrides Modula-3's structural equivalence for this type. See the language reference for more information about branding.

Next, we declare an array of title names. The outside world, of course, does not know about the existence of this array as it is not visible from the interface `Person`.

```
CONST
   Title = ARRAY Gender OF TEXT {"Ms.", "Mr."};
```

Procedure `FullName` is the implementation of method `fullname()` of `Person.T`. Since `FullName` is not exported by the `Person` interface it will not be visible to any outside modules.

```
PROCEDURE FullName (p: T): TEXT =
BEGIN
   RETURN Title[p.gender] & " " & p.firstname &
      " " & p.lastname;
END FullName;
```

The next procedure, `Describe`, is exported and hence visible to all clients of the `Person` interface. Since `Describe` is defined within this module, the representation of `p.fullname()` is visible within its body.

```
PROCEDURE Describe(p: T; desc: TEXT; wr: Wr.T := NIL) =
BEGIN
   IO.Put (p.fullname() & " is " & desc & ".\n", wr);
END Describe;

BEGIN
END Person.
```

5.4.2 Clients of an Opaque Type

`OpaqueExample` is a client of the `Person` interface:

```
MODULE OpaqueExample EXPORTS Main;
IMPORT Person;
```

There, we assign new `Person.T` instances to three variables `jane`, `june`, and `john` (using various combinations of positional and keyword parameter binding.)

```
VAR
   jane: Person.T :=
      Person.New(firstname := "June",
                 lastname := "Doe",
                 gender := Person.Gender.Female);
   june := Person.New("June","Doe",
                      gender := Person.Gender.Female);
   john := Person.New("John","Doe", Person.Gender.Male);
```

Next, we call `Person.Describe` a few times. Note that `jane.firstname`, `jane.lastname`, or even `june.fullname()` are not available in this module, even though they are available within the implementation of `Person`, since `Person.T` is an opaque type.

```
CONST
   address = "123 Main Street";
BEGIN
   Person.Describe (jane, "lives at " & address);
   Person.Describe (june, "lives at " & address);
   Person.Describe (john, "lives at " & address);
```

Indeed, if you were to invent your own `Person` type—even if its structure was the same as that of `Person.T`—the compiler would prevent you from passing the imposter into `Person.Describe`. The only way to get a new `Person.T` object is by calling `Person.New`.

```
END OpaqueExample.
```

By using opaque types, the `Person` interface achieves full encapsulation.

Sometimes full encapsulation is too strong. In the rest of this section, you learn how to encapsulate only parts of your objects.

5.4.3 Partially Opaque Types: Revealing Types in Moderation

Opaque types hide and reveal type information in an extreme manner. If the concrete implementation of an opaque type is revealed in your current scope, you know everything about the structure and the implementation of the type. If the concrete revelation is not available in your scope, then you may know nothing about the structure and implementation for the type.

In practice, you may need more control over the visibility of types in different parts of your programs. *Partial revelation* of opaque types enables fine-grained control over the visibility of fields and methods of your objects.

A natural extension of the opaque type concept, partially opaque types may be used to generalize the language-enforced visibility rules. Using partial revelation, you may define visibility rules that fit your particular application, instead of confining your object types to the hard-coded public, private, protected, and friend visibility rules common in other languages.

A new version of the `Person` interface illustrates partial revelation. A partially opaque type `Person.T` is defined, with two operations `init` and `fullname`.

```
INTERFACE Person;
```

Using the idiom `TYPE T <: Public; Public = OBJECT ... END`, the next fragment states that the `Person.T` supports at least operations `init` and `fullname`, without revealing the exact structure of `Person.T`, or revealing what other methods `Person.T` may support.

To declare `Person.T`, first, we declare a type `T` as a subtype of `Public`.

```
TYPE
   T <: Public;
```

Then, we define `Public`, the publicly available revelation of this `Person.T` to be an object type, with the methods `init` and `fullname`.

```
Public = OBJECT
   METHODS
      init(firstname, lastname: TEXT; gender: Gender): T;
      fullname(): TEXT;
END;
```

The name Public is used here by convention, not by a hard-coded rule. The method init initializes the object using firstname, lastname, and gender and returns the initialized object. The init method is used by convention to initialize values as they are.

The method fullname returns the full name of the Person.T object in question.

```
    Gender = {Female, Male};
END Person.
```

In contrast with fully opaque definition of the person interface, there is no need to provide a New procedure in the interface. Clients of this interface can freely instantiate Person.T using the built-in NEW operation. Another difference is that clients of a partially-opaque type can invoke methods on it. In this case, the methods init and fullname are available to all clients of Person. After calling a built-in NEW operation, you can call init to initialize the newly instantiated object. The method fullname returns a text string containing the name of a person; hence, there is no need for the Describe procedure to exist inside this module.

Declaring a partially opaque type also allows clients of this interface to create new subtypes of Person.T. By calling Person.T.init, such subtypes can assure that the Person.T portion of the object is initialized properly.

```
MODULE Person;
```

The implementation of the Person interface implements the partially opaque type Person.T. To do so, it *reveals* the representation of Person.T fully.

The Person interface already defines the signatures for procedures init and fullname. It is the role of this module to implement these methods, and add the underlying implementation structure. The BRANDED keyword will ensure that instances of other types with identical structure cannot masquerade as Person.T objects.

```
REVEAL
   T = Public BRANDED OBJECT
      firstname, lastname: TEXT;
      gender: Gender;
   OVERRIDES
      init := Init;
      fullname:= FullName;
   END;
```

Procedure Init initializes a Person.T object, and returns the self parameter just initialized.

```
PROCEDURE Init (self: T; firstname, lastname: TEXT;
                gender: Gender): T =
BEGIN
   self.firstname := firstname;
   self.lastname := lastname;
   self.gender := gender;
   RETURN self;
END Init;
```

Define the procedure `FullName`, the implementation of method `fullname()` of `Person.T`. Procedure `FullName` itself is not exported to the clients of `Person` interface, but the method `fullname()` of `Person.T` is visible to clients.

```
PROCEDURE FullName (p: T): TEXT =
BEGIN
   RETURN Title[p.gender] & " " & p.firstname & " " &
      p.lastname;
END FullName;

CONST
   Title = ARRAY Gender OF TEXT {"Ms.", "Mr."};
BEGIN
END Person.
```

5.4.4 Subtyping Partially Opaque Type

In the next module, `Employee.T` is defined as a subtype of a partially opaque type `Person.T`.

```
INTERFACE Employee;
IMPORT Person;

TYPE
   T <: Public;
   Public = Person.T OBJECT
   METHODS
      init(first, last: TEXT;
           gender: Person.Gender;
           company: TEXT): T;
   END;
END Employee.
```

The declaration of `Employee.T` is similar to that of `Person.T`. In this case, only a new method is declared. (As you will see, `Employee.T` also overrides the implementation of `Person.T.fullname()` method, however, `Employee`'s clients need not know this. So, if the implementation of `fullname()` changes, `Employee`'s clients don't need to be re-compiled.)

`Employee.Public` defines the publicly available definition of `Employee.T`, to be used in full revelation of `Employee.T` inside `Employee`'s implementation.

```
MODULE Employee;
IMPORT Person;
IMPORT TextIntTbl, Fmt;

REVEAL
   T = Public BRANDED OBJECT
      company: TEXT;
      id: INTEGER;
   OVERRIDES
      init := Init;
      fullname := FullName;
   END;
```

The above statement gives the full implementation of `Employee.T`, along with its fields `company`, and `id`, and its method implementations. Note that `Employee.Public` is just a shorthand for `Person.T OBJECT METHODS init(...) END`; it is named `Public` only for convenience.

```
VAR
   employee_count := NEW(TextIntTbl.Default).init();
```

`Init` defines the implementation of the `Employee.T.init` method. It takes an extra `company` parameter.

```
PROCEDURE Init (self: T;
                first, last: TEXT;
                gender: Person.Gender;
                company: TEXT): T =
VAR
   emp_id := 0;
BEGIN
   EVAL Person.T.init(self, first, last, gender);
   self.company := company;
   EVAL employee_count.get(company, emp_id);
   INC(emp_id);
   self.id := emp_id;
   EVAL employee_count.put(company, emp_id);
   RETURN self;
END Init;
```

`Fullname` defines the implementation of the `Employee.T.fullname` method. Note how it uses its supertype's `fullname` method by calling `Person.T.fullname(self, ...)`.

```
PROCEDURE FullName(self: T): TEXT =
BEGIN
   RETURN Person.T.fullname(self) & ", " &
      "employee #" & Fmt.Int(self.id) &
      " at " & self.company & ",";
END FullName;
```

```
BEGIN
END Employee.
```

5.4.5 Clients of a Partially Opaque Type

The main module, `PartiallyOpaque`, imports both `Person` and `Employee`.

```
MODULE PartiallyOpaque EXPORTS Main;
IMPORT Person, Employee;
IMPORT IO;
```

The next procedure, `Describe`, uses the `fullname` method of `Person.T` to print a textual description of a person. Note that `Person.T` does not reveal its internal structure, but it does reveal the `fullname` method, which is enough to allow the `Describe` procedure to be included this module instead of the `Person` module.

Instead of having to change `Person` every time a client requires a new `Describe` procedure, the new structure allows each client to implement its own `Describe` procedures without affecting `Person`.

```
PROCEDURE Describe(p: Person.T; desc: TEXT) =
BEGIN
   IO.Put (p.fullname() & " is " & desc & ".\n");
END Describe;
```

The next statements assign new `Person.T` instances to four variables `john`, `jane`, `june`, and `jack`. Note the use of the `v := NEW(T).init(...)` idiom in declaring, instantiating, and initializing these instances.

```
VAR
   john := NEW(Person.T).init("John", "Doe",
                              Person.Gender.Male);
   jane := NEW(Employee.T).init("Jane", "Doe",
                                Person.Gender.Female,
                                "ACME Ltd");
   june := NEW(Employee.T).init("June", "Doe",
                                Person.Gender.Female,
                                "Mass. State");
   jack := NEW(Employee.T).init("Jack", "Smith",
                                Person.Gender.Male,
                                "ACME Ltd");
```

Now, assign a new `Person.T` with a special `fullname` method to the variable `madonna`.

```
VAR
   madonna := NEW(Person.T, fullname := Madonna);

PROCEDURE Madonna(<*UNUSED*>self: Person.T): TEXT =
BEGIN
   RETURN "Madonna"
END Madonna;
```

The main body will make a few calls to `Describe`. Note the use of `john.fullname()` to call a method defined on a `Person.T`, and `Person.T.fullname(june)` to call a `Person.T` method on an `Employee.T`. Of course, the latter case is type-checked at compile-time.

```
CONST
   address = "123 Main Street";

BEGIN
   Describe (madonna, "a pop icon");
   Describe (john, "a resident at " & address);
   Describe (jane, "a resident at " & address);
   Describe (june, "a resident at " & address);
   Describe (jack, john.fullname() & "'s brother.");
   Describe (june, "called " &
      Person.T.fullname(june) & " outside work");
END PartiallyOpaque.
```

Partially opaque types are powerful structuring constructs for building large programs. For more information on partially opaque types, see the language specification, and **I/O Streams: Abstract Types, Real Programs** in *Systems Programming with Modula-3*.

5.5 Generics: Resuable Data Structures and Algorithms

A *generic* is a template for instantiating similar modules. Generics—called parameterized types, or templates in other languages— allow you to build generic data structure and algorithm code and readily use them in different contexts. For example, a generic hash table module could be instantiated to produce tables of integers, tables of text strings, or tables of a user-defined type. Different generic instances are compiled independently: the source program for the generic and its parameters is reused, but the compiled code for one instance has no relationship with other compiled instances.

To keep Modula-3 generics simple, they are confined to the module level: generic procedures and types do not exist in isolation, and generic parameters must be entire interfaces. In the same spirit of simplicity, there is no separate type checking associated

with generics. Implementations are expected to expand the generic and type-check the result.

Usually generic interfaces and modules contain code that operates independently of the type of data it operates on. The type that the generic will be operating upon is defined at compile-time.

5.5.1 Using Generics

In this example, we use the standard `List` generic interface to implement a set abstraction, and the standard `Table` generic interface to implement a mapping from names to action procedures.

Before exploring the program, let's review the `List`, `Atom`, and `Atom-List` interfaces. An abbreviated version of each interface is included here; see the on-line version of the interface for full comments.

5.5.2 A Generic Example: List

The generic interface `List` provides operations on linked lists of arbitrary element types.

```
GENERIC INTERFACE List(Elem);
```

Where `Elem.T` is not an open array type and the `Elem` interface contains:

```
CONST Brand = <text-constant>;
PROCEDURE Equal(k1, k2: Elem.T): BOOLEAN;
```

`Brand` must be a text constant. It will be used to construct a brand for any generic types instantiated with the `List` interface.

```
CONST Brand = "(List " & Elem.Brand & ")";
```

A `List.T` represents a linked list of items of type `Elem.T`.

```
   TYPE T = OBJECT head: Elem.T; tail: T END;
   PROCEDURE Cons(READONLY head: Elem.T; tail: T): T;
   PROCEDURE List1(READONLY e1: Elem.T): T;
   PROCEDURE List2(READONLY e1, e2: Elem.T): T;
   PROCEDURE List3(READONLY e1, e2, e3: Elem.T): T;
   PROCEDURE FromArray(READONLY e: ARRAY OF Elem.T): T;
   PROCEDURE Length(l: T): CARDINAL;
   PROCEDURE Nth(l: T; n: CARDINAL): Elem.T;
   PROCEDURE Member(l: T; READONLY e: Elem.T): BOOLEAN;
   PROCEDURE Append(l1: T; l2: T): T;
   PROCEDURE AppendD(l1: T; l2: T): T;
   PROCEDURE Reverse(l: T): T;
   PROCEDURE ReverseD(l: T): T;
END List.
```

5.5.3 Parameter to a Generic: Atom

Interface `Atom` will be used as a parameter to the `List` generic interface, hence `Atom` must include a `Brand` constant and an `Equal` comparison procedure.

```
INTERFACE Atom;
```

An `Atom.T` is a unique representative for a set of equal texts (like a Lisp atomic symbol.)

```
   TYPE T <: REFANY;
   CONST Brand = "Atom-1.0";
   PROCEDURE FromText(t: TEXT): T;
   PROCEDURE ToText(a: T): TEXT;
   PROCEDURE Equal(a1, a2: T): BOOLEAN;
   PROCEDURE Hash(a: T): INTEGER;
   PROCEDURE Compare(a1, a2: T): [-1..1];
END Atom.
```

5.5.4 Instantiating a Generic: AtomList

Finally, we instantiate a `List` with an `Atom` parameter:

```
INTERFACE AtomList = List (Atom) END AtomList.
```

`AtomList` can be imported by other modules that use lists of atoms. The main program for this example imports `AtomList`.

Most generic interfaces have been pre-instantiated for common datatypes such as `Text`s and `Atom`s. Indeed, `AtomList` is one such interface. See the *Interface Index* for an overview of the available generic interfaces. The language tutorial and reference manual also describe the behavior of generics in more detail.

The rest of this section describes how you can use instantiated generics, and how you can instantiate generics with user-defined parameters.

5.5.5 Using Instances of Generics

The main module, `Generics`, uses an `AtomList` to keep track of names, as well as an instance of the `Table` interface that maps `Atom`s to `Actions`. The instantiation of the table with a user-defined type `Action` is described later. Import the `Action` interface, defined in this package, and the `AtomActionTbl`, a table mapping atoms to actions.

```
MODULE Generics EXPORTS Main;
IMPORT Atom, AtomList;
IMPORT Action, AtomActionTbl;
IMPORT Process, IO; <* FATAL IO.Error *>
```

Atom List operations. Insert an element into the list.

```
PROCEDURE Insert (VAR list: AtomList.T;
                  atom: Atom.T) =
BEGIN
   IF NOT AtomList.Member (list, atom) THEN
      list := AtomList.Cons (atom, list);
   END
END Insert;
```

Print all elements of the list by iterating over its members.

```
PROCEDURE Print(x: AtomList.T) =
BEGIN
   WHILE x # NIL DO
      IO.Put (Atom.ToText (x.head) & " ");
      x := x.tail;
   END;
END Print;
```

Command operations. Actions define initial values for the action table.

```
CONST
   Actions = ARRAY OF Action.T {
      Action.T { "show", Show},
      Action.T { "quit", Quit},
      Action.T { "reset", Reset},
      Action.T { "help", Help}};
```

Each procedure defines what each action should do. Note that the `proc` field of `Action.T` is defined to be a `PROCEDURE()`, so we can assign any of `Quit`, `Rest`, `Help`, or `Show` to fields of `Actions`.

```
PROCEDURE Quit() = BEGIN Process.Exit(0); END Quit;
PROCEDURE Reset() = BEGIN input_set := NIL; END Reset;

PROCEDURE Show() =
BEGIN
   Print(input_set); IO.Put ("\n");
END Show;

PROCEDURE Help() =
BEGIN
   IO.Put("Commands: show, reset, help, or quit.\n" &
      "Other items will be inserted into the list.\n");
END Help;
```

The variable `command_table` is an atom→action table; `input_set` is an atom list, containing all the elements that will be entered.

```
VAR
   command_table := NEW(AtomActionTbl.Default).init();
   input_set : AtomList.T := NIL;
BEGIN
   FOR x := FIRST(Actions) TO LAST(Actions) DO
      EVAL command_table.put(Atom.FromText (x.name), x);
   END;

   IO.Put ("Welcome to the atomic database.\n");
   IO.Put ("Try any of commands: show quit reset help.\n");
   IO.Put ("Any other string will be entered into the" &
      "database.\n\n");
```

Loop, get the command line. If it's a command, do it. Otherwise insert the command line into the `input_set`. If `atom` is in the `command_table` then run the corresponding `Action.T`; otherwise, `Insert` the `atom` into the `input_set`.

```
   LOOP
      IO.Put ("atom-db > ");
      IF IO.EOF () THEN EXIT END;
      VAR
         cmd := IO.GetLine();
         atom := Atom.FromText(cmd);
         action: Action.T;
      BEGIN
         IF command_table.get(atom, action)
         THEN action();
         ELSE Insert(input_set, atom);
         END;
      END;
   END;
END Generics.
```

See the *Interface Index* for more information on various kinds of generics.

5.5.6 Instantiating Generics for User-Defined Types

Since `Action` is a user-defined type, there are no pre-instantiated interfaces available for it. CM3-IDE provides handy makefile procedures for instantiating various generics automatically.

The Table interface requires a *key* and a *value* parameter:

```
GENERIC INTERFACE Table (Key, Value) = … END Table.
```

The `Key` in this case is `Atom`, the `Value` is an `Action`. Here, `Action` is a user-defined interface. `Action.T`, denoting actions for commands, is a procedure with no parameters and no results. `Action.Brand` is used by the table generic to create a composite brand for our table.

```
INTERFACE Action;
TYPE
   T = RECORD
      name : TEXT
      handler : PROCEDURE();
   END
CONST
   Brand = "Action";
END Action.
```

5.5.7 Instantiating Generics in a Makefile

Finally, the makefile for this package will instantiate an atom→action table with the name `AtomActionTbl`:

```
import ("libm3")
table("AtomAction", "Atom", "Action")
module("Action")
implementation ("Generics")
program ("atom-db")
```

(If you haven't built your package yet, you won't be able to see the contents of `AtomActionTbl` because it is generated as part of the build process.)

5.6 Unsafe Constructs: System Programming in CM3-IDE

This program illustrates the use of unsafe constructs, such as `LOOPHOLE`—an unsafe cast.

The default mode for programs in CM3-IDE is *safe*, i.e., the language and its runtime are responsible for checking run-time errors. For programming intricate systems, integrating legacy systems, or making programs more efficient, you may decide that you would like the freedom to perform tasks that circumvent language-enforced safety.

You have the freedom to perform unsafe operations in *unsafe* modules by using additional operations, such as `LOOPHOLE` (an unsafe cast to an arbitrary type) or `ADR` (address of a variable). These operations are restricted to unsafe modules because they violate invariants enforced and assumed by the language in the safe modules.

With the freedom in unsafe modules comes the responsibility for the programmer to check for run-time errors in place of the language runtime. *You* are now responsible for making sure that a `LOOPHOLE` is not causing run-time errors.

Separation of safe and unsafe codes is a key technique in writing portable programs that utilize unsafe or non-portable features of particular systems. Indeed it is common practice for systems programmers to divide their code into safe and unsafe portions,

even if the programs are written in C. This way, the bulk of porting to a new platform, lies in the unsafe portion. CM3-IDE extends this model by providing language support for separation of safe and unsafe modules. Both interfaces and modules can be marked as UNSAFE.

If you care about robustness of your code, you are best to code most (if not all) of your programs in the (default) safe mode, since it is much easier to understand and explain the behavior of safe programs, hence it is also easier to make them robust.

A safe module can only import safe interfaces, so in safe programming you can't mistakenly count on unsafe functionality in another unsafe module.

An unsafe module can make its functionality available to other safe modules by exporting a safe interface. This is how you can bridge the safety gap—otherwise if your program includes one unsafe module, then your whole program must be marked unsafe. When you export a safe interface from an unsafe module, you the programmer are guaranteeing the intrinsic safety of the calls in the safe interface.

One nice aspect of the inclusion of the unsafe features in the language is that you don't have to rely on calls to external, lower-level languages to make your programs more efficient. Indeed, the unsafe portions of your code will have as much control over the representation and layout of your data structures as you have when programming in an unsafe language like C. The support for unsafe modules has been used to implement operating systems, windowing systems, networking software, and the language run-time itself in Modula-3, a task that is not easily accomplished with other high-level languages.

5.6.1 Unsafe Coding Example

In this small example, an interface to the standard C library call `abs` is marked inside an unsafe interface `Clib`, which is imported by an unsafe main program, `UnsafeExample`. Note that `UnsafeExample` cannot be marked safe because it imports an unsafe interface.

```
UNSAFE INTERFACE Clib;
<*EXTERNAL*> PROCEDURE abs(x: INTEGER): INTEGER;
END Clib.
```

The pragma `<*EXTERNAL*>` declares a procedure to be provided at link-time by external code. (In this case, by the C runtime.)

Following, the main module imports `Clib` and uses `Clib.abs`.

```
UNSAFE MODULE UnsafeExample EXPORTS Main;
FROM Clib IMPORT abs;
IMPORT IO, Fmt;

CONST
   an_integer = 10;
BEGIN

   IO.Put ("Absolute value of ");
   IO.PutInt (an_integer);
   IO.Put (" is ");
   IO.PutInt (abs(an_integer));
   IO.Put (".\n");

   IO.Put ("Absolute value of " & Fmt.Int(-an_integer) &
      " is " & Fmt.Int(abs(-an_integer)) & ".\n");
END UnsafeExample.
```

As an application programmer using CM3-IDE, you need not worry about unsafe modules unless you are writing code where efficiency is the first concern, or you are using non-portable features of an operating system or external component. The full details of the available unsafe features are described in the *Language Reference*.

5.7 Summary

Exceptions. A robust program must handle error conditions well. Exceptions are a convenient language construct for handling errors and abnormal conditions in a way that preserves your program structure.

Objects. A flexible structuring construct for building programs, objects in CM3-IDE are garbage-collected. They conform to a single-inheritance hierarchy.

Threads. Often you may need to manage multiple concurrent activities as part of your program. Threads provide the required features.

See Andrew Birrell's *Introduction to Programming with Threads*, in the Technical Notes section of your distribution, for a thorough introduction to multi-threaded programming.

Opaque types. For full encapsulation of the internal structure of types in one module from other modules, you can use opaque types. Opaque type visibilities are enforced by the language.

Partially Opaque Types. To allow multiple levels of visibility of objects in your programs, you can use partially opaque types. They are a generalization of hard-coded visibility rules such as public, private, protected, and friend modes in other languages.

Generics. Reuse is an important aspect of large program development. Known also as templates or parametrized types, generic interfaces and modules allow you to reuse data structures and algorithms. See **CM3-IDE Interface Index** on page 143 for a list of available generics.

Unsafe Programming. While safe programming is the default for CM3-IDE, at times you may not be able to avoid the need for unsafe code, for example, to write efficiency-critical portions of your program, or to interface with other languages. Unsafe modules allow you to perform tasks that are ordinarily disallowed by the language safety semantics. CM3-IDE provides mechanisms for managing unsafe portions of your code and for combining unsafe code with safe code.

This page left blank intentionally.