Olivetti in 1989 for systems programming. (See the text box entitled "History of Modula-3.") The designers had several goals:

- To provide the abstractions necessary to structure large systems programs: modules, objects, threads, and generics.
- To provide the mechanisms for making programs safe and robust: strong type checking, exceptions, isolation of unsafe code, and automatic garbage collection.
- To keep the language simple. Features were chosen that had been proven in other languages, but compatibility with older languages wasn't important.

The result was a full-featured language for software engineering and object-oriented programming. A feature-by-feature comparison puts Modula-3 roughly on par with Ada and C++; see Table 1. However, Modula-3 avoids the complexity of those larger languages by simplifying individual features. For example, Modula-3 supports object-oriented programming but implements single rather than multiple inheritance. It supports generics, but the mechanism is considerably simpler than that of Ada or C++. In practice, these simplifications do not affect day-to-day programming. Paradoxically, Modula-3 is also the most stable language: C++, Ada, and Modula-2 are all being "enhanced" in standards committees, in many cases to add features already found in Modula-3.

Table 1: Feature comparison of some popular programming languages.

	Modula-3	C++	Ada	Mo

Generics	yes	no *	yes			
Exceptions	yes	no *	yes	no *	no	no
Threads	yes	no	yes	no	no	no
OOP	yes	yes	no *	no	yes	no
User-defined operators	no	yes	yes			
Interfaces	yes	no	yes			
Strong typing	yes	some	yes			
Runtime safety checks	yes	no	yes			
Isolate unsafe features	yes	no	yes			
Procedure types	yes	yes	no			
Case-sensitive names	yes	yes	no			
Garbage collection	yes	no	no			

* These features are coming in new versions of the language.

SRC Modula-3

Before discussing language features, I should note that DEC provides free-of-charge a high-quality Modula-3 compiler, called SRC Modula-3, which is available in source form on the Internet (gatekeeper.dec.com in directory /pub/DEC/Modula-3). SRC Modula-3 runs on most UNIX workstations and is in use at many universities, companies, and research laboratories. SRC Modula-3 also includes a rich runtime library, including UNIX and X-window interfaces, and an object-oriented X-Window programming system called Trestle.

Touring the Language

Modula-3's syntax holds no big surprises; it is based on Modula-2 and, therefore, Pascal. Statements, expressions, and declarations are similar to those found in other Pascal-family languages. Modula-3, however, deviates from Modula-2 when necessary. For example, the precedence of arithmetic and logical operators follows the more natural convention found in C, Ada, and Fortran rather than the one used in Pascal and Modula-2; see Table 2.

Table 2: Some differences between Modula-2 and Modula-3.

	Modula-3	
Declarations	Names visible throughout scope; can initialize variables.	V t i
Types	Structural equivalence.	Name equivalence.
Expressions	C-like precedence; A.B shorthand for A^.B, and so on.	Pascal-like precedence; no shorthand for A^.B.
Statements	FOR loop declares its own variable.	FOR-loop variable must be declared by programmer.
Pointers	Syntax: REF T or REFANY; runtime type testing and garbage collection.	Syntax: POINTER TO T; no runtime type testing; manual deallocation of storage.
Built-in functions	MIN, MAX apply to value pairs, yield smaller and larger values.	MIN, MAX apply to types, yield smallest and largest elements.
Strings	Variable-length, read-only; different from ARRAY OF CHAR.	Fixed maximum length, read-write, same as ARRAY OF CHAR.
Isolation of unsafe features	UNSAFE keyword reveals unsafe features of language.	Most unsafe features are provided by SYSTEM interface.
OOP, generic, exceptions	Supported.	Not currently available.

In large programs, it is important to place some structure on collections of procedures and variables, restricting the proliferation of names. Modula-3 programs are structured as collections of modules and interfaces. An interface specifies a set a public facilities: types, variables, constants, and procedures. The interface is a contract between the facilities' developers and their clients. A module implements an interface by supplying private data and bodies for interface procedures. To use an interface, a client must import the interface. Interfaces and modules are stored in

different files and compiled separately.

You can change a module without recompiling clients of the interface. Consider the Modula-3 version of "Hello, World" in Figure 1. The Hello module exports (implements) an interface named Main, a built-in interface that identifies the starting point of a program. Hello also imports two interfaces, Wr and Stdio, that provide basic stream-oriented I/O facilities. (These interfaces are part of the standard libraries supplied with SRC Modula-3.) Wr.PutText is an example of how names of imported procedures and variables are qualified by the interface name. This makes it easy to keep track of name sources in large programs.

Figure 1: Modula-3 version of the classic "Hello, World!" program.

```
MODULE Hello EXPORTS Main;
IMPORT Wr, Stdio;
BEGIN
    Wr. PutText(Stdio.stdout, "Hello, World!\n");
    Wr. Close(Stdio.stdout);
END Hello.
```

More Details.

Field Lists

In the rest of this article I'll illustrate the features of Modula-3 by looking at a realistic example--an input line parser modeled on the input facilities of the Awk language. In most programming languages, dealing with free-form numeric and text input is a hassle. Even C, which has a pretty good I/O library, forces you to descend into the mysteries of scanf to read numbers. In Awk, input is effortless: Input lines are automatically broken into whitespace-delimited fields that are referred to by number and can be used as text or in numeric expressions. The goal is to write a module to provide Awk-like input for Modula-3.

Modula-3 supports both object-oriented and traditional programming models. In this case, our input-parser interface uses an object model in which a client creates an object called a "field list" and then uses it to read fields from input lines. The steps to follow are:

- 1. Create a field-list object, fl.
- 2. Read a line into fl by calling fl.getLine().
- 3. Get the value of the nth field as a string with fl.text(n). Get the value of the nth field as a number with fl.number(n).
- 4. When finished with the current line, go back to step 2.

The design for field lists gives us the opportunity to discuss two particularly interesting features of Modula-3: opaque types and threads. Opaque object types allow you to reveal only the user-visible part of an object definition in an interface, hiding its implementation in a module. Modula-3's support for threads lets you take advantage of preemptive multitasking on any computer, and real multiprocessing on computers that support it.

The Interface

The FieldList interface (see Listing One, page 126) declares several types, two constants, and an exception. The principal type (the field list) is named T by convention; clients will refer to it as FieldList.T. The relevant declarations in Listing One begin with the statement T<: Public; and include the METHODS ..END block. The <: operator signals that this is an opaque-type declaration. Translated into English, the declarations say, "Type T is an object type descended from type Public, which in turn is descended from type MUTEX. Type Public (and hence T) has these methods." Thus, we have the method specifications for T, but have yet to explain the private data and methods. (We'll see how type T's declaration is completed later.)

The FieldList interface also includes declarations to support error handling (the EXCEPTION declaration and the RAISES clauses in the method declarations) and multitasking (the type MUTEX in the declaration of Public and the Thread.Alerted exception in the declaration of getLine).

There are a few other interesting things to note in this interface. Some names are used before they are declared. This is because in Modula-3, the visibility of a name extends both before and after the name in the current scope. Type Rd.T is a "reader," a general input stream that acts like the input side of a C FILE * stream. Type NumberType is the particular floating-point type used to represent the numeric values of fields. Modula-3 has three floating-point types:

REAL, LONGREAL, and EXTENDED, corresponding to the IEEE floating-point standard types. Several convenient shortcuts are also demonstrated. The declaration getLine(rd: Rd. T:= NIL) RAISES {...} indicates that method getline takes a single parameter of type Rd. T, and that the parameter has a default value of NIL. You can omit the argument when calling the method. Even more concisely, the declaration init(ws:=DefaultWS): T, the parameter of the init method has a default value, and the parameter type is omitted. Modula-3 determines the types from the default value, so this declaration is the same as init(ws: SET OF CHAR: = DefaultWS): T. You can also omit the type in variable declarations if you supply an initial value of the same type.

The Client

Listing Two (page 126) is Sum, a small program that reads lines, sums all the numbers on each line, and prints the result. The basic idea is simple, but it demonstrates several features of Modula-3. The field list is created in the top-level variable declaration VARfl: = NEW (FieldList.T).init(WhiteSpace). The NEW function creates a new field-list object, to which the init method is immediately applied. The init method returns the initialized object. Modula-3 does not have automatic constructors (like C++), but use of the init method is a convention. The program calls fl.getLine() to read a line from the standard input and loops over the input fields, summing the numbers.

The loop body also shows an example of a WITH statement. Modula-3's WITH statement differs from those of Pascal and Modula-2, which are used to make record field names visible. In Modula-3, WITH is used to introduce a new identifier and bind it to an arbitrary variable or value for the duration of the enclosed statements. If the value is a variable designator (such as an array element or record field), the new name is aliased to the variable, and it can be read or written. Otherwise, as in this case, the new identifier is a read-only value. WITH is surprisingly convenient, and you will see many examples of it in the implementation of FieldList.

Exceptions

Error handling in Modula-3 programs is accomplished with exceptions. By using exceptions, you don't have to check return values on every procedure call--something so tedious that most C programmers don't bother to do it. The FieldList interface exports the exception Error, which is raised by certain methods in response to an error. A typical client error would be trying to read the 12th field in a line with only 11 fields.

When an exception is raised, it propagates out of the current procedure into the caller. If it is not handled there, it continues to propagate outward until a handler is found. If no procedure handles the exception, the Modula-3 runtime system terminates the program with a suitable message. Exceptions are part of the specification of procedures in Modula-3; every procedure or method that can raise an exception must list that exception in a RAISES clause. Note the RAISES {Error} clauses in the method declarations in FieldList.

The Sum program deals with exceptions in two ways: by handling some exceptions and ignoring others. The outer loop in the main program is surrounded by a TRY..EXCEPT..END block, which handles any exceptions raised by procedures inside the loop. This particular exception handler simply prints a message and allows the program to finish normally. One message is provided for the end-of-file exception, and another message for all other exceptions.

An alternative to providing an exception handler can be seen in the Put procedure, which includes the pragma <*FATAL Wr.Failure, Thread.Alerted*>. Because there is no exception handler or RAISES clause in the Put procedure, it is a checked runtime error to raise any exception within that procedure. But the compiler knows from the Wr interface that Wr.PutText can raise the Wr.Failure and Thread.Alerted exceptions, so it will warn the programmer of the potential runtime error. The FATAL pragma says, in effect, that it's OK to halt the program if these exceptions are raised within Put, and so the warning message should be suppressed.

The Module

It's time to turn to the implementation of FieldList, shown in listing Three, page 126. Given the interface shown in Listing One, this module must do at least two things: complete the opaque declaration of FieldList.T, and supply procedures to implement the methods for that type.

Note the REVEAL declaration in Listing Three. This "revelation" adds to the previous declaration of T in the interface a set of private fields that clients cannot see. The keyword BRANDED is required for reasons we won't go into

here. Notice also that the object fields have initializers. These initializations are performed whenever the object is created, and take the place of C++ constructors. For more complex initialization, a separate method must be used.

The revelation also associates procedures with methods by a series of lines of the form methodname:=procedure name. In the FieldList example we've given the procedures the same names as the methods. The method and its procedure must have compatible signatures. The signatures for isANumber are shown in Figure 2. The procedure includes an extra argument representing the object on which the method is operating. Some programmers name the extra parameter self by convention. The method call fl.isANumber(n) is equivalent to isANumber(fl, n), assuming that this procedure is currently bound to the method.

Figure 2: Signatures for isANumber.

```
  Method:
      isANumber (n: FieldNumber) : BOOLEAN RAISES {Error}
  Procedure:
      isANumber (self: T; n: FieldNumber) : BOOLEAN RAISES {Error}
```

In addition to providing better abstraction and information hiding, keeping the type revelation in the module means that changing the hidden fields or procedures of the object type does not force clients to be recompiled.

Strings and Arrays

You'll notice that the FieldList interface uses the built-in string type TEXT, but the module also uses arrays of characters. Strings (type TEXT) are extremely convenient in Modula-3. They are dynamically allocated and can be of any length. Once created, a TEXT value cannot be changed, so strings have value semantics--no one can change the value of a string you are holding. The built-in interface Text provides basic construction and testing operations on strings, but no searching functions.

For intense character manipulation, you can also convert a TEXT value to an array of characters, as in the getLine procedure. In most languages, arrays of characters are difficult to deal with because they must have a fixed compiletime size. Modula-3 provides a compromise: Although stack-based arrays must have a fixed size, dynamic arrays can be allocated with a runtime size. For example, the FieldList.T object contains a field declared as chars: REF ARRAY OF CHAR. This is a reference (pointer) to an open (unbounded) array of characters. In getLine, the self.chars and Text.SetChars statements store in self.chars a reference to an array of line-Length characters and then fill that array with the characters from string text. Unlike C and C++, dynamic arrays in Modula-3 have subscript bounds checking built in. The AddDescriptor procedure is a good example of how to use dynamic arrays, including how to grow them when necessary. All open arrays are 0 based, and the function NUMBER(a) can be used to determine the number of elements in any array a. SUBARRAY can be used to designate a contiguous set of elements in an array.

Threads

Multitasking with threads is a useful structuring tool in many applications. Threads are independent control points, or mini processes, that execute within your program's address space. Each thread has its own stack but shares access to global variables and the heap. It is not unreasonable for a large program to have dozens of threads performing various activities. Most new operating systems, including Windows NT, OS/2, and Mach (OSF/1) provide built-in support for preemptive multitasking with threads. So does Modula-3, even on operating systems that don't provide thread support directly.

Coupled with the benefits of threads is the danger of race conditions, which occurs when two threads attempt to modify a shared data structure at the same time. One thread may be suspended after partially modifying the data structure, leaving the data structure in an inconsistent state. A second thread might then trip over the inconsistency. The solution is to synchronize access by "locking" the data structure while it is being used, using a mutual-exclusion semaphore ("mutex" for short). Each thread locks the mutex while using the data structure; if a thread already has the mutex locked, the second thread will be forced to wait.

The implementation of FieldList does not use threads, but it is "thread friendly." That means FieldList provides the necessary synchronization so that clients can be multithreaded without worrying about race conditions. In Modula-3, mutexes are provided by the built-in object type MUTEX. You can store MUTEX objects in your data structures or, as in FieldList, you can simply make your object type a descendant of MUTEX, effectively turning your object into a mutex itself. In using mutexes, there is a special block

statement, LOCK mu DO..END, so that you can't forget to unlock a locked mutex. The LOCK statement locks the mutex mu while the enclosed statements are executed, and ensures that the mutex is properly unlocked when the statements terminate, even if they are terminated by an exception or RETURN within the LOCK statement. Throughout the FieldList module, you will see LOCK statements surrounding access to field lists.

Safety

What is Modula-3's most important advantage? Without a doubt, it is safety. While the exception mechanism encourages the creation of robust programs, the Modula-3 language is inherently safe and does not require special attention by the programmer.

Some of the hardest bugs to find are those that cause a valid source-code statement to execute incorrectly at runtime. For example, the C statement s->a[i]=v could fail for many reasons: The pointer s might be null or might point to unallocated storage; the value of i might be too large to use as an index into the array; or v might contain an out-of-bounds value because it was never initialized. ANSI C lists 97 ways in which a C program's behavior might be unpredictable at compile time or run time. Even Ada, usually considered a "safe" language, does not protect you against dangling pointers or uninitialized variables.

Modula-3 guarantees that all runtime assumptions remain valid. It will initialize your variables (if you don't) to ensure that they always contain values of their declared types. It checks pointer conversions at runtime for type safety, and does not permit you to deallocate storage directly. Some features found in other languages, such as taking the address of a local variable, are prohibited because they are unsafe and because detecting their misuse at run time would be too costly. These checks and rules avoid many bugs and catch the remaining ones quickly, before their effects can spread. In my experience, the error messages provided by the Modula-3 runtime system are so good that I rarely have to use the debugger to locate the cause of an error.

A good example of this safety and the new programming features it makes possible is runtime type testing. Consider the getReal procedure in Figure 3, which accepts a pointer of any type (type REFANY) and returns as a floating-point number the value pointed to.

Figure 3: Procedure to accept a pointer of any type and return as a floating-point number the value pointed to.

```
  PROCEDURE GetReal(ptr: REFANY) : REAL = (* Return ptr^ as a REAL *)
      VAR realPtr:= NARROW(ptr, REF REAL);
      BEGIN
          RETURN realPtr^;
      END GetReal;
```

The built-in NARROW function is used here to convert a "pointer to anything" to a "pointer to REAL" (REF REAL). In most languages, this code would be unsafe: The argument ptr might point to some other type of value whose bits don't constitute a valid floating-point number. Modula-3, however, checks at run time that ptr is a value of type REF REAL. If it is not, the call to NARROW will fail. This runtime type checking is possible because all dynamic storage contains type information primarily intended for the garbage collector. You can use this type information yourself. For example, Figure 4 is another version of GetReal that shows how runtime type testing can be made explicit.

Figure 4: Making explicit runtime type testing in the GetReal procedure.

```
  PROCEDURE GetReal2(ptr: REFANY) : REAL = (* Return prt^, or 0.0*)
      BEGIN
          IF ptr # NIL AND ISTYPE(ptr, REF REAL) THEN
              RETURN NARROW(ptr, REF REAL)^;
          ELSE
              RETURN 0.0; (* ptr is not what we expected *)
          END;
       END GetReal2;
```

Some programmers worry about garbage-collection overhead, but what are the real costs and benefits? Good garbage-collection algorithms seem to impose no more than a 10 percent overhead on runtime performance, and can actually save time on smaller programs or programs that use inefficient heap managers. This is a reasonable investment for reliability. Garbage collection also shortens development and makes programs smaller by eliminating the need to write storage-management code. Many OOP languages, such as Small-Talk, Eiffel, and Trellis include garbage collection.

Loopholes

Of course, systems programmers cannot restrict themselves to "safe" programming at all times. Languages that are too strict about safety may not be usable for writing low-level code, like storage allocators. Therefore, if you declare a module UNSAFE, Modula-3 gives you access to a variety of unsafe but practical features, such as unrestricted type conversions (via function LOOPHOLE), address arithmetic, and the DISPOSE procedure to deallocate storage. The compiler cannot ensure the safety of UNSAFE modules, but at least it makes you isolate and identify unsafe code.

The Tools

A programming language without good tools is almost useless. As previously mentioned, SRC Modula-3 includes a rich runtime library, including UNIX and X-window interfaces and an object-oriented X-Window programming system. Program rebuilding is easy in SRC Modula-3 using the supplied m3 driver program. The command line m3 -o Progmake * .i3 * .m3 will cause the driver to inspect all the interfaces (*.i3) and modules (*.m3) in the current directory; compute dependencies based on IMPORT and EXPORT declarations; determine which source files have changed and which interfaces are out of date; recompile anything that needs to be recompiled; and link program Prog. For more complex programs, the m3make utility allows you to describe your program abstractly without computing file dependencies by hand. You can also add arbitrary make-like dependencies to your m3makefile.

Modula-3 brings together the long-term maintainability of Ada, the simplicity of Pascal, and the modern object-oriented programming facilities of C++. The result is a clean language that makes it easy to write robust and maintainable programs.

References

Harbison, Samuel P. Modula-3. Englewood Cliffs, NJ: Prentice Hall, 1992.

Modula-3 News. Pittsburgh, PA: Pine Creek Software.

Nelson, Greg. Systems Programming with Modula-3. Englewood Cliffs, NJ: Prentice Hall, 1991.

Usenet News Group: comp.lang. modula3.

History of Modula-3

Modula-3 was developed by researchers at DEC's Sytems Research Center (SRC) and the Olivetti Research Center in 1989. It borrows from two evolutionary lines of programming languages: an academic line, represented by Niklaus Wirth's Pascal, Modula-2, and Oberon languages; and an industrial research line, represented by the Mesa, Cedar, and Euclid languages from the Xerox Palo Alto Research Center. Its immediate parent is Modula-2+, an extension of Modula-2 developed at SRC in the early 1980s and used in their research systems. In 1986, Maurice Wilkes, who had developed the first practical electronic stored-program computer at Cambridge 37 years earlier, sparked an effort to "clean up" Modula-2+. With Niklaus Wirth's blessing, this became a design for a new language, Modula-3. The original language report was issued in 1988, with minor revisions in 1989 and 1990.

Modula-3 emphasizes safety and maintainability and is gaining important converts outside DEC. The Computer Science Laboratory at Xerox PARC has adopted the language for its research software, and the University of Cambridge in England is now teaching Modula-3 to its computer science students.

--S.H.

SAFE PROGRAMMING WITH MODULA-3
by Sam Harbison

[LISTING ONE]

```
<a name="0234_001a">

INTERFACE FieldList;
(* Breaks text lines into a list of fields which can be treated
 as text or numbers. This interface is thread-safe. *)
IMPORT Rd, Wr, Thread;
EXCEPTION Error;
CONST
   DefaultWS = SET OF CHAR{' ', '\t', '\n', '\f', ','};
```

```
    Zero: NumberType = 0.0D0;
TYPE
    FieldNumber = [0..LAST(INTEGER)]; (* Fields are numbered 0, 1, ... *)
    NumberType = LONGREAL; (* Type of field as floating-point number *)
    T <: Public; (* A field list *)
    Public = MUTEX OBJECT (* The visible part of a field list *)
    METHODS
        init(ws := DefaultWS): T;
            (* Define whitespace characters. *)
        getLine(rd: Rd.T := NIL)
           RAISES {Rd.EndOfFile, Rd.Failure, Thread.Alerted};
            (* Reads a line and breaks it into fields that can be
                    examined by other methods. Default reader is Stdio.stdin. *)
        numberOfFields(): CARDINAL;
            (* The number of fields in the last-read line. *)
        line(): TEXT;
            (* The entire line. *)
        isANumber(n: FieldNumber): BOOLEAN RAISES {Error};
            (* Is the field some number (either integer or real)? *)
        number(n: FieldNumber): NumberType RAISES {Error};
            (* The field's floating-poinnt value *)
        text(n: FieldNumber): TEXT RAISES {Error};
            (* The field's text value *)
    END;
END FieldList.
```

<a name="0234_001b">
<a name="0234_001c">

[LISTING TWO]

<a name="0234_001c">

```
MODULE Sum EXPORTS Main; (* Reads lines of numbers and prints their sums. *)
IMPORT FieldList, Wr, Stdio, Fmt, Rd, Thread;
CONST WhiteSpace = FieldList.DefaultWS + SET OF CHAR{','};
VAR
    sum: FieldList.NumberType;
    fl  := NEW(FieldList.T).init(ws := WhiteSpace);
PROCEDURE Put(t: TEXT) =
    <*FATAL Wr.Failure, Thread.Alerted*>
    BEGIN
        Wr.PutText(Stdio.stdout, t);
        Wr.Flush (Stdio.stdout);
    END Put;
BEGIN
    TRY
        LOOP
            Put("Type some numbers: ");
            fl.getLine();
            sum := FieldList.Zero;
            WITH nFields = fl.numberOfFields() DO
                FOR f := 0 TO nFields - 1 DO
                    IF fl.isANumber(f) THEN
                        sum := sum + fl.number(f);
                    END;
                END;
                WITH sumText = Fmt.LongReal(FLOAT(sum, LONGREAL)) DO
                    Put("The sum is " & sumText & ".\n");
                END(*WITH*);
            END(*WITH*);
        END(*LOOP*)
    EXCEPT
        Rd.EndOfFile =>
            Put("Done.\n");
        ELSE
            Put("Unknown exception; quit.\n");
    END(*TRY*);
END Sum.
```

<a name="0234_001d">
<a name="0234_001e">

[LISTING THREE]

<a name="0234_001e">

```
MODULE FieldList;
(* Designed for ease of programming, not efficiency. We don't bother to reuse
   data structures; we allocate new ones each time a line is read. *)
IMPORT Rd, Wr, Text, Stdio, Fmt, Thread, Scan;
CONST DefaultFields = 20;       (* How many fields we expect at first. *)
TYPE
    DescriptorArray = REF ARRAY OF FieldDescriptor;
```

```
    FieldDescriptor = RECORD
        (* Description of a single field. The 'text' field and 'real'
           fields are invalid until field's value is first requested.
           (Invalid is signaled by 'text' being NIL. *)
        start  : CARDINAL   := 0; (* start of field in line *)
        len    : CARDINAL   := 0; (* length of field *)
        numeric: BOOLEAN    := FALSE; (* Does field contain number? *)
        text   : TEXT       := NIL; (* The field text *)
        number : NumberType := 0.0D0; (* The field as a real. *)
    END;
REVEAL
    T = Public BRANDED OBJECT
        originalLine: TEXT; (* the original input line *)
        chars      : REF ARRAY OF CHAR := NIL; (* copy of input line *)
        nFields    : CARDINAL := 0; (* number of fields found *)
        fds   : DescriptorArray := NIL; (* descriptor for each field *)
        ws    : SET OF CHAR := DefaultWS; (* our whitespace *)
    OVERRIDES  (* supply real procedures for the methods *)
        init           := init;
        getLine        := getLine;
        numberOfFields := numberOfFields;
        line           := line;
        isANumber      := isANumber;
        number         := number;
        text           := text;
    END;
PROCEDURE AddDescriptor(t: T; READONLY fd: FieldDescriptor) =
    (* Increment the number of fields, and store fd as the
    descriptor for the new field. Extend the fd array if necessary. *)
    BEGIN
        IF t.nFields >= NUMBER(t.fds^) THEN
           WITH
              n  = NUMBER(t.fds^), (* current length; will double it *)
              new = NEW(DescriptorArray, 2 * n)
           DO
              SUBARRAY(new^, 0, n) := t.fds^;   (* copy in old data *)
            t.fds := new;
           END;
        END;
        t.fds[t.nFields] := fd;
        INC(t.nFields);
    END AddDescriptor;
PROCEDURE getLine(self: T; rd: Rd.T := NIL)
    RAISES {Rd.EndOfFile, Rd.Failure, Thread.Alerted} =
    (* Read an input line; store it in the object; finds all the
    whitespace-terminated fields. *)
    VAR
        next      : CARDINAL; (* index of next char in line *)
        len       : CARDINAL; (* # of characters in current field *)
        lineLength: CARDINAL; (* length of input line *)
    BEGIN
        IF rd = NIL THEN rd := Stdio.stdin; END;  (* default reader *)
        LOCK self DO
            WITH text = Rd.GetLine(rd) DO
             lineLength         := Text.Length(text);
             self.originalLine := text;
             self.fds        := NEW(DescriptorArray, DefaultFields);
             self.nFields    := 0;
             self.chars      := NEW(REF ARRAY OF CHAR, lineLength);
             Text.SetChars(self.chars^, text);
            END;
            next := 0;
            WHILE next < lineLength DO (* for each field *)
              (* Skip whitespace characters *)
                   WHILE next < lineLength AND (self.chars[next] IN
                                                self.ws) DO INC(next);
                END;
                (* Collect next field *)
                len := 0;
                WHILE next < lineLength
                   AND NOT (self.chars[next] IN self.ws) DO
                    INC(len); INC(next);
                END;
                (* Save information about the field *)
                IF len > 0 THEN
                       AddDescriptor(self, FieldDescriptor{start:=
                                                    next - len, len := len});
                END;
            END(*WHILE*);
        END(*LOCK*);
    END getLine;
PROCEDURE GetDescriptor(t: T; n: FieldNumber): FieldDescriptor RAISES {Error} =
    (* Return the descriptor for field n. Depending on user's wishes,
    treat too-large field numbers as empty fields or as an error. *)
    BEGIN
        (* Handle bad field number first. *)
        IF n >= t.nFields THEN
            RAISE Error;
        END;
        (* Be sure text and numeric values are set. *)
        WITH fd = t.fds[n] DO
          IF fd.text # NIL THEN RETURN fd; END; (* Already done this *)
          fd.text := Text.FromChars(SUBARRAY(t.chars^, fd.start,
```

```
                                                                    fd.len));
          TRY (* to interpret field as floating-point number *)
              fd.number  := FLOAT(Scan.LongReal(fd.text), NumberType);
              fd.numeric := TRUE;
            EXCEPT
                Scan.BadFormat =>
                    TRY (* to interpret field as integer *)
                        fd.number  := FLOAT(Scan.Int(fd.text),
                                                                    NumberType);
                        fd.numeric := TRUE;
                    EXCEPT
                        Scan.BadFormat => (* not a number *)
                            fd.number  := Zero;
                            fd.numeric := FALSE;
                    END;
            END;
            RETURN fd;
        END(*WITH*);
    END GetDescriptor;
PROCEDURE numberOfFields(self: T): CARDINAL =
    BEGIN
        LOCK self DO RETURN self.nFields; END;
    END numberOfFields;
PROCEDURE isANumber(self: T; n: FieldNumber): BOOLEAN RAISES {Error} =
    BEGIN
        LOCK self DO
            WITH fd = GetDescriptor(self, n) DO RETURN fd.numeric; END;
        END;
    END isANumber;
PROCEDURE number(self: T; n: FieldNumber): NumberType RAISES {Error} =
    BEGIN
        LOCK self DO
            WITH fd = GetDescriptor(self, n) DO RETURN fd.number; END;
        END;
    END number;
PROCEDURE line(self: T): TEXT =
    BEGIN
        LOCK self DO RETURN self.originalLine; END;
    END line;
PROCEDURE text(self: T; n: FieldNumber): TEXT RAISES {Error} =
    BEGIN
        LOCK self DO
            WITH fd = GetDescriptor(self, n) DO
                RETURN self.fds[n].text;
            END;
        END(*LOCK*);
    END text;
PROCEDURE init(self: T; ws := DefaultWS): T =
    BEGIN
        LOCK self DO
            self.ws            := ws;
        END;
        RETURN self;
    END init;
BEGIN
    (* No module initialization code needed *)
END FieldList.
```

Figure 1: Modula_3 version of the classic "Hello, World!" program

```
MODULE Hello EXPORTS Main;
IMPORT Wr, Stdio;
BEGIN
    Wr.PutText(Stdio.stdout, "Hello, World!\n");
    Wr.Close(Stdio.stdout);
END Hello.
```

Figure 2. Signatures for isANumber.

```
Method  Procedure
isANumber(n: FieldNumber): BOOLEAN
 RAISES {Error} isANumber(self: T; n: FieldNumber): BOOLEAN
 RAISES {Error}
```

Figure 3. Procedure to accept a pointer of any type and return as a floating-point number the value pointed to.

```
PROCEDURE GetReal(ptr: REFANY): REAL = (* Return ptr^ as a REAL *)
    VAR realPtr := NARROW(ptr, REF REAL);
    BEGIN
        RETURN realPtr^;
    END GetReal;
```

Figure 4. Making explicit run-time type testing in the GetReal

```
procedure

PROCEDURE GetReal2(ptr: REFANY): REAL = (* Return ptr^, or 0.0 *)
   BEGIN
      IF ptr # NIL AND ISTYPE(ptr, REF REAL) THEN
         RETURN NARROW(ptr, REF REAL)^;
      ELSE
         RETURN 0.0; (* ptr is not what we expected *)
      END;
   END GetReal2;
```

Related Reading

- News
- Commentary

News

- **Parallels Supports Docker Apps**
- **Devart dbForge Studio For MySQL With Phrase Completion**
- **Restlet Completes "Complete" API Platform**
- **Docker Clocks In On Azure**

More News»

Commentary

- **Dr. Dobb's Archive**
- **Google's Data Processing Model Hardens Up**
- **Xamarin Editions of IP*Works! & Integrator**
- **Parasoft DevTest Shifts To Continuous**

More Commentary»

- Slideshow
- Video

Slideshow

- **Jolt Awards 2015: Coding Tools**
- **Developer Reading List**
- **Jolt Awards: The Best Programming Utilities**
- **C++ Reading List**

More Slideshows»

Video

- **Watson Discovery Advisor Cloud Service**
- **Intel at Mobile World Congress**
- **Research@Intel Showcase**
- **Amazon Connection: Broadband in the Rainforest**

More Videos»

- Most Popular

Most Popular

- **Lambda Expressions in Java 8**

- **Developer Reading List: The Must-Have Books for JavaScript**
- **An Algorithm for Compressing Space and Time**
- **Why Build Your Java Projects with Gradle Rather than Ant or Maven?**

More Popular»
