EFLIB

USER'S GUIDE

RELEASE 3
1999-02-28

EFLIB is a sophisticated application framework for every Pascal programmer. This book guides you through EFLIB, and teaches you how to create your own, powerful applications.

TABLE OF CONTENTS

– Where to look for what

Table of contents 2

1. What and why 4

Welcome to EFLIB! 4

Who is EFLIB for? 4

The core of EFLIB 4

Features in EFLIB 5

Required equipment 5

Installation guide 5

About this book 5

2. Object primer 7

The object 7

Inheritance 7

Polymorphism and information hiding 8

Classes and instantiation 8

Creating your first class 8

Exercises 9

2. Getting started 10

Units: the building blocks 10

Using objects 11

Construction and destruction 11
Virtual methods and VMT's 11
Dynamic memory allocation 11

The parent class 12

3. Fundamental classes 14

Kernel classes 14

Streams 14

Creating a stream 14
Using streams 15
Other streams 16

Filters 16

Encryption filter 17
Sequential filters 17

Files 18

4. Data structures 19

Introduction 19

The ADT parent classes 19

Linear ADT's 20
Ordered ADT's 20

Arrays 20

Lists 21

Composite ADTs 21

Stacks 21

Queues 21

Hash tables 22

Trees 22

How to customize elements 22

Nodes: the element packaging 22

5. Mathematics 24

Introducing a the OO approach 24

Standard arithmetics 24
Complex numbers 24
Matrixes 25
Polynomials 25
Expressions 25
Mathematical functions 25
Equation solvers 25

6. User interactions 26

Introduction to the GUI 26
Components and managers 26

7. Advanced programming 27

Extending classes 27
Constructors and destructors 27
Overriding methods 28
Registering a new class 28
Component selectors 29
Plug technology 29
Stream storage mechanism 30
Storing an instance 30
Loading an instance 30
How to support stream storage 30

A. Tables 32

Class registration identities 32

B. Error messages 33

Installation 33
Developing 34

C. Definitions 35

1. WHAT AND WHY

– A simple solution with flexible features.

Welcome to EFLIB!

Welcome to EFLIB. You are about to explore a new world of programming possibilities for the Pascal programming platform. The idea behind EFLIB is to make programming easier – both for novice users and expert programmers. EFLIB is a complete application framework that revolutionizes programming in Pascal:

- Intuitive user interface
- Event-driven programming
- Classic data structures
- Powerful mathematics
- Solid object-orientation

Who is EFLIB for?

EFLIB provides elegant object-oriented solutions. A novice programmer can easily use them without knowing very much about how EFLIB actually works beneath the surface. Novice programmers will enjoy the user interface. It will enable them to write small programs and still give them a very flexible and uniform user interface.

Furthermore, EFLIB is a great asset in education. The solid object orientation is suitable as a start platform for those who want to explore object oriented programming. The naming convention is makes it easy to learn OO programming. When you have learned a little about EFLIB, you will see that many other things work in the same manner.

EFLIB simplifies and optimizes advanced programming, and hence reduces the time needed for program developing. Moreover EFLIB makes your source code uniform and very readable. Your programs will be easier to modify and more stable.

The core of EFLIB

When designing a program, we usually concentrate on what the program is supposed to do and give little attention to how, from the perspective of the user, the program will be used. If you have a difficult problem to solve, it is natural to concentrate on the solution rather than the way the solution is packaged. The packaging is important, however, because a program that does the right thing but is difficult to use is of little or no value. EFLIB provide this packaging.

EFLIB is an application framework. Being an application framework, EFLIB highly promote extension of its classes. Most classes have special features that simplify extension. Some parts of EFLIB use a plug technology to make extension easier. This plug technology makes it possible to attach certain classes to another class. Depending on what kind of class was plugged in, it replaces some built-in feature in the extended class. For example, data structures can be told to handle element comparisons differently in several ways. The simplest way is to install a plug that overrides the built-in compare mechanism.

This and other technologies give EFLIB an unlimited extendibility. You can customize EFLIB and add new features to the framework whenever you want to.

Features in EFLIB

- EFLIB includes a complete user interface with windows, message boxes, buttons, menues, editors and much more.
- There is a complete set of classic data structures delivered with EFLIB: linked lists, vectors, tables and trees.
- Polymorphic data streams, filter classes and device classes simplifies every I/O process in EFLIB.
- EFLIB comes with a full math package, that includes vectors, matrixes and complex numbers, as well as numerical solvers and expression parsing.

Required equipment

- Borland Pascal with Objects 7.0
- MS-DOS or compatible operating system such as IBM OS/2 or Microsoft Windows.
- A good memory configuration with DPMI or much available memory in the low 640K region.
- Optionally a mouse with installed driver.

Installation guide

1. Extract the files from the EFLIB archive with the PKUNZIP tool from PKWARE® Inc. Use the -d parameter to restore the directory structure.
2. Make sure the computer has at least 600 KB free memory. If you do not have enough memory, try to optimize your memory configuration. Consult your operating system manual for further help. If you are using a DPMI-version of Borland Pascal or a compiler that can compile huge programs, the memory in the DOS-region is of no concern.
3. Start Borland Pascal.
4. Go to the pull down menu "Options" and choose "Directories".
5. Move to the field "Unit directories" and add ";C:\EFLIB" after the existing text. Replace the directory name with the name of your EFLIB unit directory, ie. the directory to which you extracted the archive in step 1.
6. Save the configuration using the choice "Save" in the "Options" pull down menu.

About this book

This guide presumes that the reader has a basic knowledge of Pascal programming. You should know a little of object-oriented programming. This guide is not intended for professional developers. Instead, they are adviced to study the comprehensive Function Reference.

The best way of learning how to use EFLIB is to combine reading with the writing of your own programs. Self-studies of the example source code that comes with the toolkit will enable to go further.

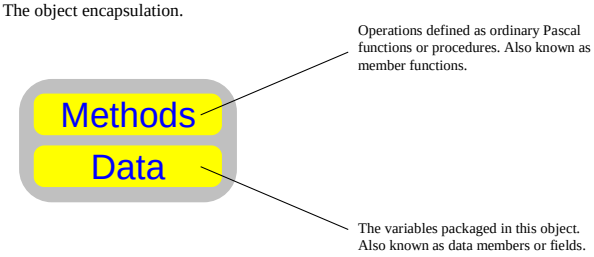
2. OBJECT PRIMER

– An introduction to object-oriented progarmming.

The object

Object orientated programming, or OOP, is a software engineering technique that views programs as a collection of components called objects that can interact. The core of OOP is the object. An object in OOP is much like a real-life object: it has properties and knows how to perform certain tasks.

Figure 2.1
The object encapsulation.



Technically, an object is a combination of data and a set of operations. Think of an object as a package of variables (the fields) and functions that only operate on the object (the methods). Objects are used in a way similar to a record data type in Pascal or a struct in C. For example, the object Car may have a data field fColor. We can access this field by writing Car.fColor.

Inheritance

Objects can inherit properties from other objects. Assume that you have drawn a sketch of a vehicle with four wheels. Inheritance would allow you to copy that sketch. You continue to paint on your copy, and soon you have a car. Another copy could become a truck, and so on. No matter how many copies you create, the original will still be there in the background, unchanged. You are not limited to make only one copy!

The concept of inheritance is very similar to the copying of a sketch. We can derive new objects from any existing object. We say that we specialize the original object into a new object. Derived objects are also known as ancestors or subclasses. A derived object automatically inherit all properties of the parent object, but we are free to add new methods or fields.

Inheritance allows the programmer to reuse objects that have been defined earlier, perhaps for different purposes. We sometimes choose to define abstract objects. An abstract object is an object that are not to be used directly. It is merely provided to define a common interface for its concrete ancestors. Since inheritance preserves the interface, all objects derived from an abstract object enable the user to call methods defined in the abstract parent object, but the implementation, the actual result of the calls, can differ. All ancestors represent the abstract object, but in different forms.

Polymorphism and information hiding

Now, assume that you want to buy a vehicle. You are more interested in that your vehicle is sufficient, than how it work. A car and a truck is both vehicles. They both have four wheels, breaks, etc. You know that your objects are sufficient and you know that they actually descend from the vehicle object, and that is all you need to know! One mechanism that enables us to use an object without knowing details about it is known as polymorphism. It is very important in OOP and often used in EFLIB.

Polymorphism is Greek and means ”many forms”. In our example, the vehicle could be a truck as well as a car. The vehicle occur in several different forms.

Polymorphism is related to dynamic binding. The compiler determines at execution time which operation you require in a particular situation. When we use our vehicle, the compiler decides if it has a car engine or a truck engine - when the program is running.

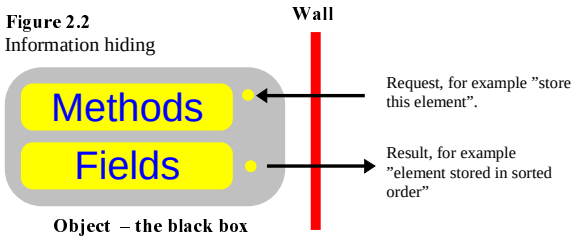


Figure 2.2
Information hiding

Polymorphism enables us to hide information about our vehicle. However, there are other ways to hide information. By using private fields and method, we can hide things that we do not wish the user to bother about. We sometimes talk about objects as black boxes or isolated modules. An object perform some well-defined tasks without letting anyone see how it works inside.

Classes and instantiation

When an object is defined using a type statement in Pascal, the resulting type is called a class. Like a record the class cannot be used until a variable have been declared. The variable is known as an instance of the class. The class is instantiated.

All classes must have at least one method called the constructor. This method initializes and creates an instance of a certain class. Constructors make it possible to configure an instance when it is created. We may for example want some fields to be zero to start with. However, there are a technical explanation to the need of constructors. Some methods are classified as virtual and need do be prepared before the instance is to be used. If you use the instance without calling the constructor first, the computer may crash. You should therefor always call a constructor before you use an instance.

Creating your first class

We have already mentioned that classes must be defined using a type statement, and that you cannot use a type directly, but must declare a variable (instance) of that type to use it. Let us create a simple object!

```
uses EFKERNEL;

{ Definition of the class }
type MyClass = object (tObject)
  constructor Initialize;
  Field : integer;
end;

{ Implementation of MyClass }
constructor MyClass.Initialize;
begin
     Inherited Initialize;
     Field := 0;
end;

{ The program }
var A : MyClass;
begin
     A.Initialize;
end.
```

Though the above example is very simple, we have actually created a new class in EFLIB. The new class (”MyClass”) inherits tObject - the base class from which all classes in EFLIB are inherited. We have added a constructor to our class. The constructor resets the member field. But before it does that, it enables the inherited class to initialize.

Exercises

1. Why is the inherited constructor called before ”MyClass” does anything?
2. Define and implement a destructor named Intercept for ”MyClass”.
3. Why should the inherited destructor be called after ”MyClass” does anything in Intercept?
4. Add a method named ”Value” that returns the value stored in ”MyClass”.

2. GETTING STARTED

– Explore the sophisticated interface in EFLIB.

Units: the building blocks

EFLIB consists of some fundamental unit files which contain the classes. Units are arranged to handle different functions. There are units for different kinds of data structures as well as for screen routines. Since you must tell the Pascal programming what units you want to incorporate with your program, it is very important that you know where EFLIBs classed are located.

EFDEF

Global definitions: constants, types and variables. All units depend on EFDEF.

EFKERNEL

The most fundamental classes in EFLIB are placed in this unit. Together these classes are called the kernel classes or simply the kernel. Most important is tObjcet, the base class from which all other EFLIB classes are derived. The kernel also manages the class hierarchy and handles run-time errors.

EFINIT

When EFLIB run on a computer system it must adjust some system specific settings. EFINIT automatically does this whenever it is incorporated with your program.

EFBASIC

Fundamental classes for timing, date handling, bit sets and coordinate managing are located in the EFBASIC unit.

EFSTREAM

EFLIB uses data streams to handle flows of data. These data streams are defined by tStream in EFKERNEL. tStream is an abstract class with concrete ancestors in the EFSTREAM unit.

EFFILTER

Flows of data can be filtered, that is processed in some way and still operate as a tStream. The interface of such filters is defined in EFFILTER. EFFILTER also supply a set of different filter classes, for instance encryption and compression.

EFIO

Fundamental file and directory operations are handled by classes in EFIO.

EFCMAN

The event-driven interface known as the Component-Manager Architecture is defined in the EFCMAN unit. This unit contains classes like tComponent and tManager.

EFDEVICE

EFDEVICE defines the general device interface in EFLIB, that is what all devices must be able to do. In EFLIB, a device is a event-driven entity capable of responding to predefined messages and to post messages whenever the device senses an action. For example, tKeyboard, a tDevice descendant located in the EFKEYBRD unit, automatically posts key-press messages when the the user presses a key on the keyboard.

EFADT

EFADT defines the general interface of EFLIBs data structure in the classes tADT, tLinearADT and tOrderedADT. EFADT also contains ADT plugs and iterator classes.

EFMATH

EFMATH contains the powerful math classes in EFLIB. These classes include complex numbers, vectors and matrixes, as well as equation solvers and expression parsers.

Using objects

In procedural programming, a unit contain some procedures and functions that you call from your programs. The object-orientation in EFLIB instead supply you with classes that you cannot use until you have declared an instance. Imagine that you want to write a program that handles a text. You then declare an instance of some class, possibly tText. To perform some operation on that text, you make calls to the methods in the class, by addressing the instance.

Construction and destruction

Instances must be constructed and initialized before they are used. This job is done when you call the constructor named Initialize, or any other constructed specific for the class you are using. When you have finished working with an instance, it must be destructed. Destruction makes it possible for the instance to restore memory it have allocated. There are two destructors that all classes must supply: Intercept and Free.

```
var MyObject : tMyClass;

  with MyObject do begin
       Initialize;
       { ... }
       Intercept;
  end;
```

There is an important difference between these destructors. Intercept releases memory used by the instance. Free calls Intercept thus releasing memory, but then it releases the memory for the instance.

Virtual methods and VMT's

Virtual methods are methods that are bound to instances at run-time rather than when you compile the program. Virtual methods are very useful. You can operate on some class without knowing what virtual method you actually are calling. You just know the syntax of that method. This is called polymorphism and discussed thoroughly in programming literature.

However, virtual methods could involve difficulties if they are misused. Virtual methods may even cause your computer to crash if you are not cautious. Since virtual methods are associated to code at run-time, it is important that the addresses to the code is right. These addresses are set up when you call the constructor in your instance. Pascal then creates a virtual memory table (known as VMT) that maps the virtual methods with their memory addresses. It is therefore extremely important that you construct an instance before you use it. Make it a custom to call the Initialize constructor as in our previous examples.

Dynamic memory allocation

Borland Pascal provides you with a flexible mechanism that makes it possible to allocate memory while the programming is running. Such memory is handled by a pointer, in our case a pointer to a class. All classes have a type definition of their corresponding pointer. tObject´s pointer type is pObject. Similarly, other classes pointer types are pStream, pADT, pList and so on. When you use dynamic memory, you cannot use a variable declared as a pointer type immediately. You must associate that pointer to some memory address; to some instance. You allocate memory for an instance by calling the New function in Pascal. When you are done with an instance, you can call the Intercept destructor, and then the Dispose function in Pascal

to release the allocated memory, or you can make a simple call to the Free destructor. It automatically invokes the Dispose function for you.

```
var MyDynamicObject : pMyClass;

  New ( MyDynamicObject, Initialize (...) );
  Dispose ( MyDynamicObject, Intercept );

  MyDynamicObject := New ( pMyClass );
  with MyDynamicObject^ do begin
       { ... }
       Free;
  end;
```

Be careful with the Free destructor. Since an instance cannot access its own pointer, your pointers are not automatically reset to NIL after you call Free. Also, notice that you cannot call the Dispose function with the Free destructor. You then ask Pascal to dispose your instances memory twice!

```
Dispose ( MyDynamicObject, Free ); { Wrong - never do this! }
```

You are free to use the same variable several times in your program (as in our example above). However, make sure that you do not make invalid calls to any destructor, for example when you have not allocated memory for an instance. Since destructors are virtual methods, such a call could cause your computer to crash.

The parent class

All classes in EFLIB descend from the same parent class: the tObject class. However, most classes are derived from classes that in their turn are derived from tObject. This gives rise to a class hierarchy – a relational network established by class inheritance.

When a class inherits another class, it automatically inherit all the properties defined in parent classes. Therefor, EFLIBs classes share common properties, properties defined in tObject. tObject do not have any data members. Most important of the common properties are:

- An object must construct when the constructor Initialize is called.
- An object must destruct when Intercept or Free is called. The latter destructor also releases memory used by the instance.
- An object knows if it is equal to another object. The method IsEqual does this.
- An object knows if the class relation to another object. Methods like IsParent, IsDerived and IsDescendant does this.
- An object knows whether or not it is compatible with another object. The method IsCompatible does this.

The following methods are important:

Methods

constructor Initialize;
Constructs an instance, that is initializes the virtual memory table and resets all data members.

destructor Intercept; virtual;
Destroys an instance, that is deallocates all internal memory and fields.

destructor Free; virtual;
Calls Intercept and then releases the memory for the instance. Notice that any instance variable is not set to NIL, and that this could become a dangerous problem.

procedure Assert (Condition : boolean; ErrorCode : word); virtual;
Asserts that the condition is TRUE. If the condition is not TRUE, a fatal error is triggered (that is, an error that must result in program termination).

procedure Error (ErrorCode : word); virtual;
Triggers the specified error. tErrorHandler is called.

function IsEqual (Instance : pObject) : boolean; virtual;
Determines if this instance is equal to the specified instance. This method can be overridden, but returns the result of a bitwise comparison by default.

function IsCompatible (Instance : pObject) : boolean; virtual;
Returns TRUE if and only if this and the specified instance are compatible. The criteria of compatibility differ between classes, but it is basically the same thing as the classes can replace each other with regard to something. A common criteria for compatibility is that two classes share a common parent class which define their interface.

3. FUNDAMENTAL CLASSES

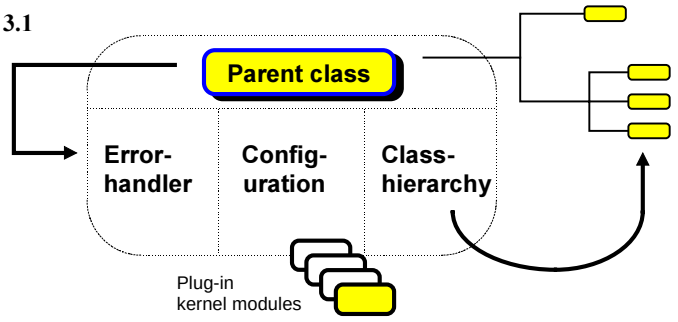
- The building blocks in EFLIB

Kernel classes

The classes in the unit EFKERNEL are called kernel classes because their function play a central role in the application framework. These classes are closely bound to each other. Most important is tObject, the parent class. This class defines the common behavior of all classes in EFLIB. tObject often request information about itself by asking questions to class hierarchy (tClassManager). If something goes wrong, the error handler is informed about the problem and decide what actions must be taken. The framework must be configured to run. The basic configuration is handled by the kernel.

The following illustration gives an overview of EFLIBs kernel:

Figure 3.1
Kernel



Streams

A data streams is as collections of data on its way somewhere: typically to a file, memory or some other device. Streams provide a simple, yet elegant, means of storing data and instances outside your program.

On a fairly fundamental level, you can think about streams much as you think about Pascal files. At its most basic, a Pascal file can be simply a sequential I/O device: you write things to it, and you read them back. A stream, then, is a polymorphic sequential I/O device, meaning that it behaves much like a sequential file, but you can read or write any data, even instances.

Streams can also (like Pascal files) be viewed as a random-access I/O device, where you seek to a position in the file, read or write at that point, return the position of the file pointer, and so on. These operations are also available with streams.

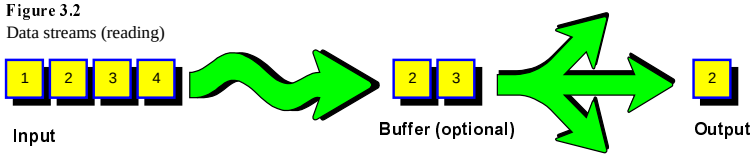
Creating a stream

The interface of data streams is defined in the class tStream in EFKERNEL, but that class is abstract and can never be used directly. Instead, you must instantiate one of the concrete stream classes in EFSTREAM, for example tFileStream. tFileStream requires that two parameters are passed to the constructor Initialize: the filename and the buffer size. The filename can be any string (e.g. ”c:\files\data.txt”). The buffer size can be any number from 0 to 65520.

```
var MyFile : pFileStream;
begin
     { tFileStream requires two parameters to construct: }
     MyFile.Initialize ('c:\files\data.txt', 1024);
end.
```

When the file stream is constructed, it allocates that memory for a buffer. Buffers give your file faster access, especially if you will make a lot of small data transfers. The principal design of buffered file streams are shown in figure 3.2.

Figure 3.2
Data streams (reading)



Using streams

All streams share a common interface. The only difference in their usage is basically the way they are constructed. tStream defines the common interface. All streams have the following important properties:

- Current position is known and returned from the Position method.
- Size or length of the stream is known and returned from the Size method.
- The size of the last transfer, that is the number of bytes that successfully could be transferred, is known and returned from the LastTransfer method.
- They provide the methods Read, Write and Seek.
- They enable instances to be loaded or stored in the Load or Store methods

Most important are the Read and Write methods:

```
MyFile.Read ( MyData, SizeOf(MyData) );
{ We now have moved SizeOf(MyData) steps forward
  in MyFile.  }
MyData [1] := 5; { Change }
{ Let us write the changed data after the unchanged! }
MyFile.Write ( MyData, SizeOf(MyData) ); { Write }
```

Both of these methods automatically moves to a new position in the stream. The resulting transfer is sometimes smaller than your request – you could have reached the end of the stream. If you are at the end, the method IsEnd returns TRUE, and your last transfer size is returned by the LastTransfer method.

```
var SourceFile, TargetFile : pFileStream; Storage : string;
begin
     New ( SourceFile, Initialize ('source.txt', 100));
     New ( TargetFile, Initialize ('target.txt', 100));

     SourceFile^.CopyOut (TargetFile, SourceFile^.Size);
```

```
     { The above CopyOut could be relplaced with this
       loop, but CopyOut is faster. }

     Storage[0] := #255;
     while not FirstStream^.IsEnd do begin
           SourceFile^.Read (Storage[1], 255);
           TargetFile^.Write (Storage[1],
                              SourceFile^.LastTransfer);
     end;

     SourceFile^.Free;
     TargetFile^.Free;
end.
```

Stream modes

Streams must always be in one of three different modes: reading, writing or both. The mode tells the stream what kind of operations it permit. The default mode is to permit both reading and writing.

When you create a file stream (tFileStream) the file you specify does not have to exist. It is automatically created if it is not found. If it exist, it is open and the stream moves to the first position. In both cases, the stream will enable both read and write access.

File streams

File streams are buffered. Whenever you read some data, the stream tries to get that data from the buffer. If the buffer is full, it must be reread. On the other hand, streams writes may require that the buffer is flushed, that is written to the file and then cleared. This two things are done in the **Flush** method.

Other streams

There are several other stream classes in EFSTREAM. Most important is tStandardStream and tNullStream. The former enables you to access the standard I/O device, that is the console and the keyboard. tStandardStream reads from the keyboard and writes to the screen. tNullStream is a blank stream. It consumes everything that it is asked to write and returns blank information instead of actually reading anything.

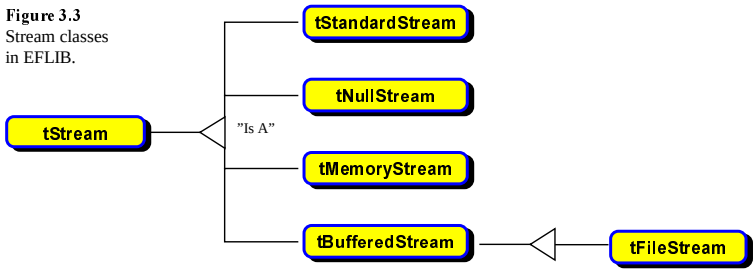


Figure 3.3
Stream classes in EFLIB.

Filters

You sometimes may want your data to be converted, but still want to provide the tStream interface. You then can use a filter. A filter is a stream that gives physical data (the base

stream) a virtual representation. For example, you may want to work with compressed information as if it was not compressed at all. You require a virtual representation that hide the actual nature of the data: it is compressed. Figure 3.4 shows how the filer works.

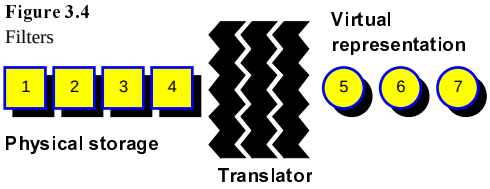


Figure 3.4
Filters

Thus, a filter is a translator with the interface of a stream. EFLIB provides you with a complete set of filters. Most important is compression and encryption.

A filter must always be associated to some other stream: the base stream. This base stream represents the physical storage, that is the raw data that we are working with. This stream must always be specified when a filter constructs. The default behavior of a filter, is to treat its base streams as a dependent instance. The filter becomes responsible for the destruction of the base stream when it is itself destructed. However, you can make the base stream independent by calling the DisableDependence method.

Encryption filter

You can encrypt any data by using tEncryptFilter. This filter uses the built-in random number generator in Pascal and controls the random seed with a keyword that you must specify.

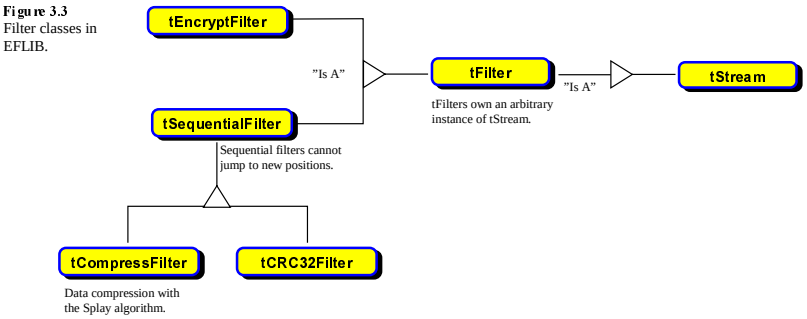


Figure 3.3
Filter classes in EFLIB.

Sequential filters

Filters derived from the class tSequentialFilter ignore calls to the Seek method. Thus, you can never change the position in the base stream other than by reading or writing data. EFLIB's compression filter is sequential. This compression filter uses a Splay algorithm to compress data when you call the Write method, and to decompress data when you call Read.

tCRC32Filter defines a filter that calculate 32-bit cyclic redundancy checks compatible with the Z-Modem transfer protocol and many programs, including PKZIP from PKWARE Inc. A CRC is a checksum that validates that your data have not been changed. tCRC32Filter is passive filter. It does not change the data, it just monitors the transfers and enables you to fetch the current CRC value at any time.

Files

In EFLIB, all files are buffered streams, that is descendants from the tBufferedStream class. A file is hence used as any other stream.

4. DATA STRUCTURES

– How to manage your information

Introduction

EFLIB comes with a complete set of classic data structures. All data structures share a common interface. This makes it very easy to use them. No matter what kind of structure you are dealing with, you always use the same terminology!

EFLIB distinguishes two categories of data structures: linear and ordered data structures. Linear data structures are structures with elements that you access simply by telling what number the element has. The first element has number 1. This number is also called the index of an element. Hence, linear data structures use a simple indexing technique. Ordered data structures are data structures that arrange their elements in some way. You therefore do not know the index of an element. Instead, all elements are associated with a search-key. You refer to an element by specifying the search key. For instance, a queue of waiting people may be known as ”A”, ”B” and ”C”. The search-key can be any information that can be calculated from the content of the element.

We have already spoken of ”elements”. What is an element? In EFLIB, an element is information managed by a certain element class derived from tElement. Data structures are not entirely responsible for their elements. Many things are handled on an individual level. Elements are responsible for themselves. The data structure merely request the element to perform some operations or to answer questions. For example, the data structure may tell its element to swap contents, to allocate some data or to compare the content with some other element.

How does this design affect your programming? Well, it becomes much more simple to customize your data! You do not have to change the data structure. All you have to do is to modify the element class. Thus, customizations of data structures are basically the same thing as extending element classes.

The ADT parent classes

All data structures are derived from the class tADT and either tLinearADT or tOrderedADT. The terminology ADT stands for Abstract Data Types.

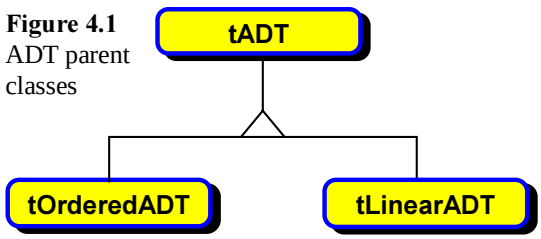


Figure 4.1 ADT parent classes

tADT define the common interface for all ADTs. tLinearADT specializes tADT into data structures that know the index of its elements. tOrderedADT instead assigns search-keys to its elements.

Linear ADT's

A linear ADT is a data structure that assigns number to its elements. These numbers are known as element index. The first element has index "1". Linear ADT's can be sorted, and is by default attached to a sort plug. A sort plug is a class that defines a way of sorting elements. The default plug uses the Quick Sort algorithm.

Ordered ADT's

Ordered data structures assign search-keys to their elements and can that way find the element you require. Ordered ADT's have exclusive access to the ordering of their elements. This mean that you cannot change the order, and specifically not sort an ordered ADT. Thus, there are no operations associated to ordered ADT's that change the ordering of their elements.

Arrays

The most simple linear data structure is the array (tArray in EFARRAY). All arrays are two-dimensional and can hold a predefined number of elements, a number known as the capacity of the array. However, an array does not have to use all its element. Arrays keep track of their elements by storing addresses to element instances in a dynamic memory allocation. Thus, the array can change places of elements by just rearranging two pointers, and the array can change its capacity or make space for new elements in the middle of the array by moving pointers, instead of the entire element data. Arrays are the only linear data structure that can hold empty elements. Basically, an array can be considered as a collections of slots. You can put one element instance in an arbitrary slot. Empty slots are marked with NIL addresses. These slots may not be accessed.

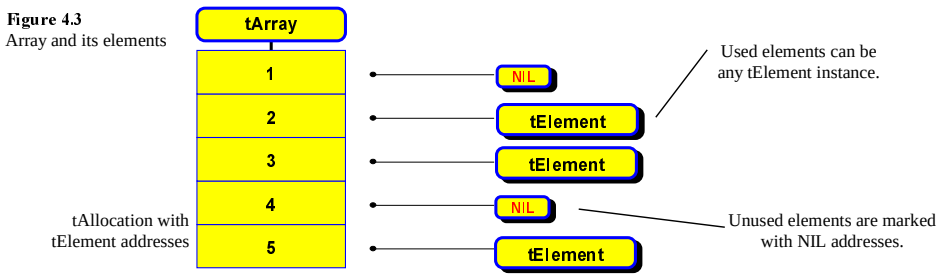


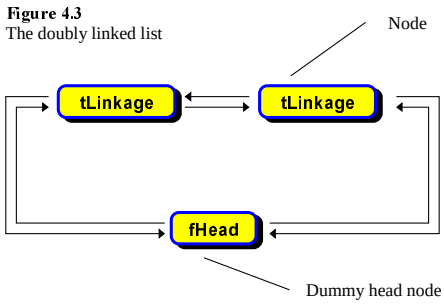
Figure 4.3
Array and its elements

Arrays know the index of the first and last used element. These indexes are returned when the methods High and Low are called. All arrays can be defragmented. This means that all used elements are put into the front and all empty slots are moved to the end.

EFLIB provides two specialized arrays: the bounded array and the virtual array. The bounded array (tBoundedArray) does not how to call the High and Low method to know the first and last used element. Instead these indexes, the boundaries, are maintained as data fields and automatically updated when the array is modified. A bounded array is automatically defragmented, that is there cannot be unused space in the middle of the array. The virtual array (tVirtualArray) inherits the bounded array, but adds the feature of automatically adjusting its size according to the demand of new elements. The array is resized in a blocks of a predefined number of elements, the growth size. Virtual arrays always have sizes that are multiples of their growth size. Like the bounded array, the virtual array is automatically defragmented. Hence, the number of used elements is always equal to (High-Low+1).

Lists

In EFLIB, all lists are doubly linked. Elements are packaged in nodes with links in both directions:



All lists have a blank head node. This head node is attached to both the first and the last element: the list is always circular. All list classes are located in the EFLIST unit. Most important is tList, the common doubly linked list. All other lists descend from this class: those are tAdjustedList, tOrderedList and tReversedList. tAdjustedList is a self-organizing data structure. When an element is used, it is automatically moved to the front of the list, hence giving it faster access next time it is used. The ordered list automatically inserts element in sorted order, and the reversed list inserts element at the end of the list. Notice that the list ADT, especially the last three lists, are somewhere in the middle of an ordered and linear data structure according to EFLIB's definition. These lists decide where elements are stored, but in the meantime, the all provide indexes for there elements.

Composite ADTs

Many ordered data structures are based on a linear data structure. So are the stack and the queue, the priority queue and the hash table. These ADTs are descendants of tOrderedADT, but they all own an instance of tLinearADT that they operate on. The interface of a composite ADT is defined in tCompositeADT. tCompositeADT define the default behavior of ordered-linear ADT composites, and provide a linear search mechanism for the Get method.

Stacks

Stacks are composite ADTs that obey the LIFO protocol (Last In - First Out). Thus, the last inserted element is the only element that can be retrieved for the stack. Stacks (tStack in EFDATA) introduce a few new methods: Push, Pop, Top and Skip. Push inserts an element into the stack. Pop retrieves the last inserted element and then erases it. Top returns the last inserted element (the top element) without erasing it. Skip throws away the current top element. Any attempt to retrieve elements from an empty stack triggers fatal errors.

Queues

Queues are composite ADTs that obey the FIFO protocol (First In - First Out). Queues are much like real-life queues. The first thing to join the queue always becomes the first one to leave it. Queues use the following methods: Enqueue, Dequeue, Skip and Circulate. Enqueue puts a new element to the back of the queue. Dequeue retrieves the element that have waited the longest time. Skip throws away the first element, and circulate moves it to the back of the queue again.

There are several types of queues, all defined in EFDATA. tQueue defines a common queue where the insertion order determines the position of an element. tCircularQueue is very similar, but Dequeue automatically moves elements to the back of the queue – elements never leave a circular queue except when they are explicitly erased. tPriorityQueue is somewhat different from the two other queues. Instead of storing elements in the insertion order, the priority queue sorts them according to their priority. The priority is determined by element comparison. The largest element is always the first one to leave the queue.

Hash tables

Hash tables (tHashTable) is basically an array, though it is implemented as a composite ADT (that is, descend from tCompositeADT). Hash tables provide very rapid access to their elements.

Trees

EFTREE contains several different tree data structures: the common tree, the binary (search) tree, the AVL tree and the splay tree.

How to customize elements

In most cases you do not have to concern yourself about tElement instances. They exist in the background and provide you with a flexibility that you benefit by without understanding the technical solution. However, tElement classes can be customized. They are customized by extension, that is by inheriting new classes from old element classes.

EFLIB provides you with a basic set of element classes. By default, all data structures use elements with the tGenericElement class. This class can store any information. It uses a dynamic memory allocation that automatically is set to the data you want to store. tGenericElement is very flexible, but uses additional memory to handle the dynamic allocation. To eliminate this memory, you have to extend tElement into classes customized for your data. EFLIB supply you with a few customized element classes that you can use straight away. Most important is tString, tCarrierElement, tStreamElement and tMathObject.

It is easy to start using other element classes. Once the class is defined, and you have created instances of that class - for example a text string with the tString class - you can easily store it to any data structure by calling the Put method. Then, the data structure takes control of that element instance. By taking control, it also is responsible for calling the Intercept destructor if the data structure is destructed. You can whenever you like to move or erase that element instance by calling the corresponding method in the data structure.

Nodes: the element packaging

Some data structures require there elements to connect to each other. This behavior is not defined in tElement. Thus, a special packaging for elements is required. This packaging is implemented in nodes that carries a tElement instance. Linked lists uses nodes with the class tLinkage. This class defines the ability to connect to two other nodes: the successor node and the predecessor node (see figure 4.3). tLinkage is derived from tContainer in EFKERNEL. The latter class define the connectivity in two directions, while the former handle the carried tElement instance. The design is illustrated in the following figure:

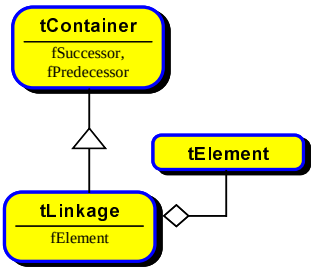


Figure 4.4
Nodes are container classes

The tree node (tTreeLinkage) is a descendant of tLinkage. This node can connect to arbitrary many other nodes, in both directions.

5. MATHEMATICS

– Powerful object-oriented mathematical routines

Introducing a the OO approach

The conventional way of implementing mathematics in Pascal programming is to define procedures that accept only a certain data type, for instance a record defining a complex number or a matrix. Thus, the border-line between different mathematical objects is very strict.

EFLIB introduces a polymorphic mathematical solution. You can now send mathematical objects to routines without regard to what kind of object you are dealing with. To start with, take a look at this example source code:

```
Z := New (pInteger, Initialize (4)); { Z = 4 }
C := New (pComplex, Initialize (2, 2)); { C = 2+2i }

with C^ do begin
     { Type-casting with complex numbers. }
     Add (Z); { Complex arithmetics with integers }
     Divide (Z);
     WriteLn ( '(C+Z)/Z = ', Re:0:0, '+', Im:0:0, 'i.' );
     Free; { pComplex }
end;
```

In the above example, we construct two numbers: an integer and a complex number. We add the integer to the complex number. Notice that the same Add method is used regardless what mathematical object we want to add to C. In our case, Z is automatically converted to the complex number 4+0i, and then added to C.

Standard arithmetics

All mathematical objects are derived from the class tMathObject in EFMATH. This class defines some abstract methods that all mathematical objects must replace. These methods are: Add, Subtract, Multiply, Divide and IsZero. All but the last require one argument: the second operand. An operation always act on the called instance. For example, C^.Add(Z) in the above example tells C to become C+Z.

Complex numbers

Complex numbers support all the standard arithmetics, but also a great deal of more advanced operations.

Matrixes

In addition to the standard arithmetics, matrixes support some specific operations. Most important is Gauss' elimination, inversion and traversal.

Polynomials

Polynomials are basically an extension of the matrix class.

Expressions

An expression is an descendant of the tMathObject with an associated instance of the class tVariables. tVariables contain the unknown variables, and can be modified to change the expression. For example, an expression could be represented as ”2+x+y” and tVariables as ”x=2” and ”y=2”. This expression would then behave like a tInteger with the value ”6”.

Mathematical functions

A mathematical function is either an expression or a pointer to some far-declared procedure that returns a function value whenever called.

Equation solvers

EFLIB contains several classes specialized in solving equations or estimating values of certain functions. Equations are solved with Newton-Raphsons method. Simpsons formula is provided for estimation of an arbitrary intergrale. Heuns and Eulers method can solve simple differential equations in two variables.

Basically, a solver is some kind of iterative process associated to some expression. The expression is defined in the tMathFunction class. The expression can either be a text string or an address to a far-declared procedure.

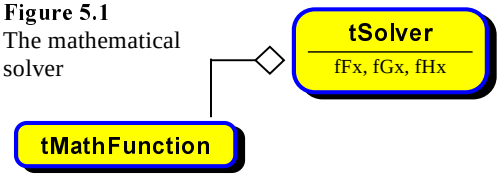


Figure 5.1
The mathematical solver

If we are using an expression, tMathFunction contain an instance of tExpression, the expression class.

6. USER INTERACTIONS

– How to make your program easy to use

Introduction to the GUI

EFLIBs graphical user interface (GUI) is event-driven. This means that different components communicate with each other using small messages. For instance. when a mouse button is clicked, a message is sent to the concerned window.

Components and managers

Let us examine the user interface in Microsoft Windows. There, a program is described in terms of an application and a window. The application contains a window. Each window can hold (contain) several other sub-windows. This is a kind of client-server architecture, since the application acts like a server to a window client, and each window can be a server for potential sub-windows.

In Microsoft Windows, a window is both a component (a client) and a manager (a server). This is not true in EFLIB. A window in EFLIB is always a component, but it can hold a manager – a tManager instance. This tManager instance enables the window to contain sub-windows. Without the tManager extension, a window can only handle itself.

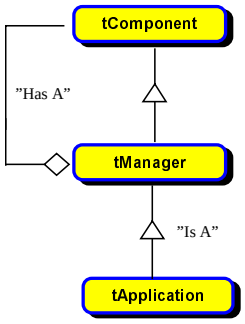


Figure 6.1. The C-M class hierarchy.

7. ADVANCED PROGRAMMING

– What is going on beneath the surface?

Extending classes

You may want to extend some classes, that is customize them according to your own needs. When you extend a class you make use of class inheritance and override some virtual methods to change the way the class work.

```
type tExtendedList = object (tList)
       constructor Initialize; { Always supply a constructor! }
       procedure Put (Element : pElement); virtual;
      end;

constructor tExtendedList.Initialize;
begin
     { Begin by calling the inherited constructor to
       asure that the instance inherited interface
       is operable. }
     Inherited Initialize;
end;

procedure tExtendedList.Put (Element : pElement);
begin
     { ... }
     Inherited Put (Element);
end;

begin
end.
```

In the above example, we do not provide a new destructor since there is no more things to take care of when the extended class destructors. Unlike constructors, destructors are virtual and does not always have to be replaced when a class is extended.

Constructors and destructors

Use inheritance to derive new classes from existing EFLIB classes. You automatically get access to all public methods and data members. You must always supply a constructor with a new class since constructors are specific for each class. When you write your constructor, always begin with a call to the inherited constructor, like:

```
Inherited Initialize ( ... );
```

Similarly, all destructor methods should end with a call to the inherited destructor, like:

```
Inherited Intercept;
```

By calling the inherited methods, you are guaranteed to have a operable instance. The inherited method prepare the inherited interface, while you must add functions that initialize new data members, etc.

There is no rule that says you must have just one constructor. Many objects in EFLIB uses several constructor to allow the user to construct objects with different properties. For example, the mathematical matrix (tMatrix) can be initialized with the Initialize constructor or with the InitializeUnitary method. The first of these requires two parameters; the number of rows and columns that is desired for the created matrix. The second of these requires only one, since it creates a square unitary matrix.

Overriding methods

If your new class is a descendant from a data type, a stream or similar, you may wish to override some virtual or non-virtual methods to change the way the data type or stream operates. Then, you have got to remember the difference between virtual and static (non-virtual) methods. Virtual methods are attached to an instance at run-time but static methods are attached at compile-time. This means that if you override a virtual method, you can be sure that your the inherited methods in your new object calls the new method instead of the old one. If you override a static method, your it will not change the behavior of methods that calls the inherited version of this method, since it's address is determined at compile-time.

It is simple to override methods. Just supply new methods with the same name as the method you want to replace, and make sure that their syntax is the same. If your inherited method is virtual, so must your new method be. Notice that only virtual methods alter the internal behavior of classes, and that virtual methods are the only way to change polymorphic interfaces. For example, the Put method in tADT is virtual, since any ancestor class then can provide a new method that is automatically called from any tADT of that class.

Registering a new class

Since the built-in class mechanism in Pascal is very limited, EFLIB uses a special class hierarchy (tClassManager) to handle classes and there relations. EFLIB keeps track of from where your class was inherited, what subclasses it have, its name, its virtual memory table, and optionally what methods should be invoked when the class is loaded or stored to a stream.

The tObject base class automatically communicate with the class hierarchy. It request information about itself whenever it need to. It is therefore recommended that you register all new classes you create, and in some cases you must do it to enable your new class to communicate with existing EFLIB components.

If you register the class you automatically enable a set of methods defined in tObject. The method NameOfType returns a string containing the full type name of the instances class, for example "tObject". An instance knows its relation to other instances. You can ask an instance if it is derived from any other instance, and vice versa. This is done in methods like IsParent, IsDerived and IsDescendant.

You register new classes by calling the method Register in the tClassManager instance named Classes provided by EFKERNEL:

```
{ Register your new class to the class hierarchy: }
Classes^.Register ( 1000, { Identity number }
  'tMyNewClass', TypeOf(tMyNewClass), TypeOf(tMyParentClass),
  @tMyNewClass.StreamLoad, @tMyNewClass.StreamStore );
```

The first parameter is a specific identity number with range like a word integer. Each new class you define will need its own, unique identity number. EFLIB reserves the registration numbers 0 through 999 for its own classes, so your registration numbers can be anything from 1000 through 65,535. It's your responsibility to keep a record of what identities you use and what classes you associate them with. The second parameter is the a text string containing the exact type name of this object, and the third is address to the VMT of the class. It must be specified and valid. The next parameter is the VMT address of the registered parent class. It may be NIL if there is no parent class. The last two parameters are used for the stream storage mechanism. It's described in detail below. The first of these two parameters is the address to the load constructor, and the second to the virtual store method. Both of these can be NIL pointers if you class does not support stream storage. If you attempt to store a class that have not been registered, your program will terminate and report an error.

tClassManager automatically distinguishes between VMT addresses and instance addresses; except at registration. When you register a class, you must specify the VMT address by calling the Pascal TypeOf function. An instance address is not accurate and will trigger a fatal error. If you program terminate and report something like "ambiguous identities" and refers to tClassManager, you have most likely an invalid class registration. Verify that your identity is not already in use, and that all parent classes are registered before their descendants are registered.

Component selectors

EFLIB uses a special type of methods to provide classes with initialized instances of other classes. These methods are called component selectors since they select a component (a class to instantiate). ADT's make use of this technology. They supply methods like CreateIterator and CreateElement. They return instances of tIterator or tElement respectively. You can easily change the behavior of an ADT by overriding a component selector method and thus force the ADT to use some other instance.

Plug technology

EFLIB also uses a plug mechanism to provide ADT´s with a modular interface for sorting algorithms and element compare techniques. This technology is defined in EFPLUG. Basically, the plug technology gives the ADT a plug manager. This manager is responsible for providing the owner of the manager (in this case, the ADT) with plugs (tPlug), and for only permitting one plug for a certain task. Ambiguous plugs are not allowed. For example, there can only be one sorting algorithm associated with an ADT. Plugs that are associated to the same task are called compatible. This compatibility is determined by the IsCompatible method in the plugs. When a new plug is installed, any compatible plug is automatically destructed. Hence, the unambiguity of the plugs are preserved.

You are not recommended to extend the plug manager, but you are encouraged to create new plugs that replace built-in plugs – for example new sorting algorithms (derived from tSortPlug in EFADT).

Stream storage mechanism

Streams provide a mechanism for object storage, that is you can store any instance of EFLIBs classes to a stream and later retrieve it, even in other applications. For example, you could store a data structure or a window to an encrypted file. This file may operate as your resource file to make your program easier to modify.

There are several benefits with stream storage:

- Memory saving technology; store your data in a file instead of putting it all in EXE file.
- Makes it easy to modify your program; let's say that you design a program that shall be used internationally. Then, you could use a file with the messages and just replace it when you need other languages for the message text (such as Spanish).

The concept of stream storage will be explained in two steps. First, we will show how an existing instance can be loaded or stored by calling certain methods in the concerned stream. After this, we will study how you make create new classes that can be stored to streams.

You call one of these tStream methods if you want to load or store an instance:

```
procedure Store (Instance : pObject);
procedure Load (var Instance : pObject);
```

Storing an instance

The store method requires an argument with the address to your existing instance. However, you can pass NIL instances to Store. NIL instances are stored as empty entries. They are loaded as NIL instances and not constructed. The tStream.Store method does the following:

- Identifies the class of the specified instance by calling tClassManager in EFKERNEL.
- Writes the identity word to the stream.
- Calls the objects StreamStore method to transfer the data inside the instance to the stream.

Loading an instance

When you have a stream with some stored instances, you can load an instance from that stream. You then make a call to the tStream.Load method. This method does the following:

- Reads the identity word from the stream.
- Identifies the class.
- Constructs and allocates memory for an instance of this type. The construction is performed with the StreamLoad method in the corresponding object type. The StreamLoad constructor transfers the data from the stream into the instance and initializes it.

The load constructor requires an argument with the variable you want to assign the loaded instance to. Notice that the load method allocates the instance. Thus, you can pass any unused pointer to Load.

How to support stream storage

Both methods require that your class is registered to tClassManager and that it support stream storage, that is it is registered together with addresses to a StreamLoad and a StreamStore method. Almost all classes in EFLIB are registered and can be stored to a stream. The only

exceptions are those that are abstract or cannot support stream storage of some technical reason (for instance, because they are streams themselves).

If you design new classes, you must register them to tClassManager and supply at least an overridden StreamLoad constructor (since constructors are specific for a certain class – you cannot use the inherited constructor because that would construct the inherited class, not the new class).

An instance can be constructed and read from a stream using the StreamLoad constructor. It require one argument: a pointer to the stream from which the instance shall be loaded.

```
constructor MyObject.StreamLoad (Stream : pStream);
begin
     Inherited StreamLoad (Stream);
     { Load the data }
     Stream^.Read (MyField, SizeOf(MyField));
     { Do not forget to initialize variables, etc. }
end;
```

An instance can be stored (written) to a stream using the StreamStore method. Like the StreamLoad constructor, StreamStore require an argument: a stream pointer.

```
procedure MyObject.StreamStore (Stream : pStream);
begin
     Inherited StreamStore (Stream);
     { Store the data }
     Stream^.Write (MyField, SizeOf(MyField));
end;
```

Notice that both methods invokes the inherited method for the same task. This is handy, since the inherited object loads or stores all previously defined fields for the object. Also, notice that the above extracts for StreamLoad and StreamStore methods does not perform a stream security check. If a NIL pointer is passed as an argument your program may crash.

A. TABLES

– A shortcut for professional programmers

Class registration identities

Classes	Identity range	Classes	Identity range
ADT base class	500	**Kernel classes**	0 .. 19
ADT plug extensions	700 .. 729	**Linear ADTs**	501 .. 599
ADT sort plug extensions	730 .. 749	**Mathematical objects**	300 .. 349
Application engine	950 .. 999	**Mathematical parsers**	380 .. 399
Basic and IO classes	220 .. 229	**Mathematical solvers**	350 .. 379
Device classes	200 .. 219	**Memory handlers**	50 .. 54
Elements	750 .. 799	**Message classes**	230 .. 249
Fundamental classes	20 .. 49	**Ordered ADTs**	600 .. 699
Fundamental C-M classes	210 .. 219	**Register classes**	260 .. 269
Fundamental plug classes	250 .. 259	**Streams and filters**	55 .. 99
GUI styles	890 .. 899	**Text and string handling**	100 .. 149
GUI	860 .. 889	**View classes**	800 .. 859

B. ERROR MESSAGES

– A guide to errors related to Pascal and EFLIB

Installation

EFLIB requires a great deal of system memory to compile. The most common error is due to a bad memory configuration.

File not found (EFxxx.TPx)
The compiler cannot find a component that is required for EFLIB to compile. The most likely cause is that you have not correctly entered the EFLIB kernel directory name in the Options Directories Unit dialogue box.

File not found (xxxxx.INC)
Verify that the Option Directories Include Files dialogue box contains the EFLIB kernel directory name.

Out of memory
Borland Pascal cannot compile or run your program since there is not enough free system memory available. Try to reconfigurate your system and remove unnecessary memory hogging programs such as TSRs. If it does not help try the following: ■ Consider installing a DPMI memory manager and the use of the Borland Pascal DPMI IDE (start with BP.EXE). ■ Set the Compile Destination to disk. Your program will then be built directly to the disk and you save some memory. ■ Set the Options Linker to disk. ■ Set the Options Debugger settings so "stand-alone" and "integrated" are disabled. ■ Edit the file FLAGS.INC and remove the line with the "$DEBUG" directive. You then disable range-checking and other debug options that waste memory. ■ If all else fails, compile your program from the DOS command-line using the "BPC" command-line compiler that is delivered with Borland Pascal.

File not found (xxxxx.INC)
Verify that the Option Directories Include Files dialogue box contains the EFLIB kernel directory name.

Unit file format error ...
Your unit files are damaged. Try to rebuild them with the BUILD.PAS program that is delivered with EFLIB (in the Kernel directory).

Developing

Most run-time errors are automatically detected and handled by EFLIB. EFLIB provides a very safe developing platform. By setting the DEBUG flag in FLAGS.INC you tells EFLIB that you want maximum protection from fatal errors such as invalid pointer operations. EFLIB will avoid computer crashes, and instead tell you that you passed an invalid pointer.

[Component name]: [error message]
EFLIB's internal error handler registered a run-time error and halted the program execution. Your program makes an invalid call to an EFLIB component.

C. DEFINITIONS

– Definitions of object-orientated concepts

Abstraction — The technique in problem solving in which details are grouped into a single common concept. [Budd93]

Abstract — A class that is not used to make direct instances, but class rather is used only as a base from which other classes inherit. [Budd93]

Class — An abstract description of the data and behavior of a collection of similar objects. Classes provide a mechanism for encapsulation and instance generation.

Constructor — A method that initializes an instance. A constructor must be called prior to instance use.

Component selector method — A method designed to provide the class with initialized instances of another class. Component selectors enable the user to change the behavior of a class by changing the components it uses. Example: CreateIterator.

Component selector class — A class exclusively dedicated to provide other classes with initialized instances of some other classes - the components.

Derived — A class (instance) X is derived from Y, if Y is any parent to X - not necessary a immediate parent. In EFLIB, X is also classified as derived from Y if X and Y is the same class.

Descendant — A class (instance) X descend from a class Y if Y is the immediate parent to X. In EFLIB, X is also classified as an descendant if X is the same class as Y.

Destructor — A method invoked when an instance no longer shall be used - when it is destroyed.

Dynamic memory allocation	A process where memory managed and handled during program execution. The programmer is responsible for the handling of dynamically allocated instances and variables.
Equal type	Instances have equal type if their classes are the same.
Extension	Where a class is inherited and some methods are overridden or added, we say that the class is extended.
Inheritance	If a class X has all the properties of a class Y, and some more, then X is inherited from Y.
Message	Some information carried between components. In EFLIB, messages are instances of a special message class.
Node	Any encapsulation of an element that enables the element to connect to other elements.
Override	The action that occurs when a method in a subclass with the same name as a method in a parent class takes precedence over the method in the superclass. [Budd93]
Parent	A class X is the parent to Y if Y is derived from X.
Reciever	The component to which a message is sent.
Virtual method	A method assigned to a class upon run-time using dynamic (late) binding.

© Johan Larsson, 1992 - 1997. All rights reserved.