Eiffel: An Advanced Introduction

Alan A. Snyder
Brian N. Vetter

Table of Contents

Introduction - Welcome To Eiffel

1 - Basic Concepts and Syntax
- 1.0 Classes and Objects
  - 1.1 References
  - 1.2.1 Declaring Expanded Objects
- 1.2 Expanded Types
- 1.3 Features
  - 1.3.1 Attributes - Variables and Constants
    - 1.3.1.1 Unique Attributes
    - 1.3.1.2 Constants
  - 1.3.2 Routines - Procedures and Functions
    - 1.3.2.1 Creation Routines
  - 1.3.3 Local attributes

2 - Inheritance, Using Objects, Clients and Suppliers
- 2.1 Inheriting from Other Classes
- 2.2 Using an Object - the Dot (.) Operator
- 2.3 Clients and Suppliers
- 2.4 Export Status
  - 2.4.1 Multiple Exports

3 - Copying, Cloning and Equality
- 3.0 Introduction
- 3.1 Copying Objects
  - 3.1.1 Shallow Copy
  - 3.1.2 Deep copy
- 3.2 Cloning Objects - Deep and Shallow
- 3.3 Object Equality - Deep and Shallow

4 - Control Flow Constructs - Conditional, Iteration, and Multi-Branch
- 4.1 Looping
- 4.2 Conditional Constructs
- 4.3 Multi-Branch

5 - Inheritance and Feature Adaptation
- 5.0 Introduction
- 5.1 Selecting
- 5.2 Renaming
- 5.3 Redefining
  - 5.3.1 Feature Signature
- 5.4 Changing the Export Status of Inherited Features

5.5 Undefinition
5.5.1 Semantic Constraints on Undefinition
5.5.1.1 Frozen Features

6 - Genericity
6.1 Unconstrained Genericity
6.1.1 Multiple Paramaterization
6.2 Constrained Genericity

7 - Using Eiffel: Examples and Tutorials
7.0 Overview
7.1 Class VEHICLE
7.2 Contract Programming
7.2.1 Preconditions
7.2.2 Post conditions
7.2.2.1 Old Expressions
7.2.3 Invariants
7.2.3.1 Looping Revisited - Variants and Invariants
7.2.4 A Note on Efficiency
7.3 The New VEHICLE class
7.4 Exceptions
7.4.1 Handling Exceptions - Rescue clauses
7.4.2 Retry Commands
7.5 LAND_VEHICLE and WATER_VEHICLE
7.6 Repeated Inheritance
7.7 HYDRO_LAND_VEHICLE

8 - Eiffel's Role in Analysis and Design and Final Words Concerning Eiffel's Future
8.0 Introduction
8.1 Deferring Classes and Features
8.2 Explicit Creation
8.3 What Do I Tell My Clients?
8.4 The Future of Eiffel

Introduction
Welcome to Eiffel

Eiffel is a pure object-oriented programming language designed with the explicit intent to produce high quality, reliable software. It is pure in its nature in that **every** entity is an object declared of an explicitly stated class type. It adheres to some of the long proven and time tested programming practices that have made languages like Modula-2 a successful engineering advancement.

Eiffel promises to be the next step toward a better understanding and more importantly, efficient use of the object-oriented practices that have evolved thus far. This paper intends to familiarize the reader with Eiffel's concepts and its sometimes unique approach to constructs. This is not intended to be an introduction to the object-oriented programming paradigm in general. Familiarity with programming (procedural, hybrid object-oriented/procedural, or pure object-oriented) is very helpful, as the material presented is in such a form that certain concepts must bee understood on the reader's part prior to reading this.

Without any more delay, we present the Eiffel programming language...

Chapter 1:

Basic Concepts and Syntax

1.0 Classes and Objects
1.1 References
1.2.1 Declaring Expanded Objects
1.2 Expanded Types
1.3 Features
1.3.1 Attributes - Variables and Constants
1.3.1.1 Unique Attributes
1.3.1.2 Constants
1.3.2 Routines - Procedures and Functions
1.3.2.1 Creation Routines
1.3.3 Local attributes

1.0 Classes and Objects

A class is essentially an extremely modular abstract data type template through which objects are produced. That is, a class is a structure with a set of operations (routines) and data entities associated with it. In Eiffel, every entity is an instance of a class- an object. Even the INTEGERs, and REALs that come as part of the standard data types are actual classes. This means that one could possibly inherit from an INTEGER, and make a new class, NEW_INTEGER if so desired.

The difference between a class and an object is similar to that of a type and a variable in traditional languages. There are two types of objects that exist in Eiffel: references and expanded objects. These may be thought of a a "pointer to" an object, and the object itself, respectively. We will now examine these objects more closely.

1.1 References

The default type of objects declared in Eiffel are references (essentially pointers without the associated dangers). This is accomplished by invoking a creation routine on the new object, as the following example demonstrates:

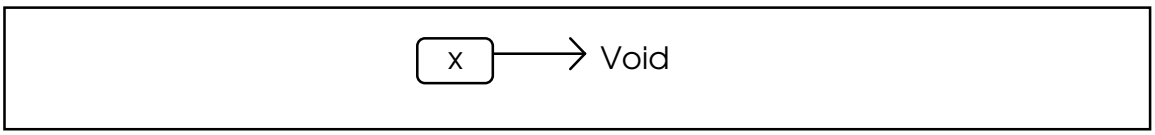
```
x : SOMECLASS;
!!x.make;  -- call the a SOMECLASS creation routine
x.some_routine;   -- some_routine is a member of SOMECLASS
```

First, *x* is declared as an instance of SOMECLASS. Entity declarations in Eiffel are similar to most procedural languages: an identifier followed by a colon (:) followed by a class name.

The next line uses the double-exclamation (!!) operator on *x* to instantiate it (allocate space for its existence). *make* is a creation routine defined in SOMECLASS that performs object initialization (its declaration is not shown here). Additionally, it may also take other actions to ready the object for use. In the *make* routine, no explicit call by the routine can be made to allocate space for *x*. The !! operator takes care of making sure that space is made available while the *make* routine can perform initialization safely.

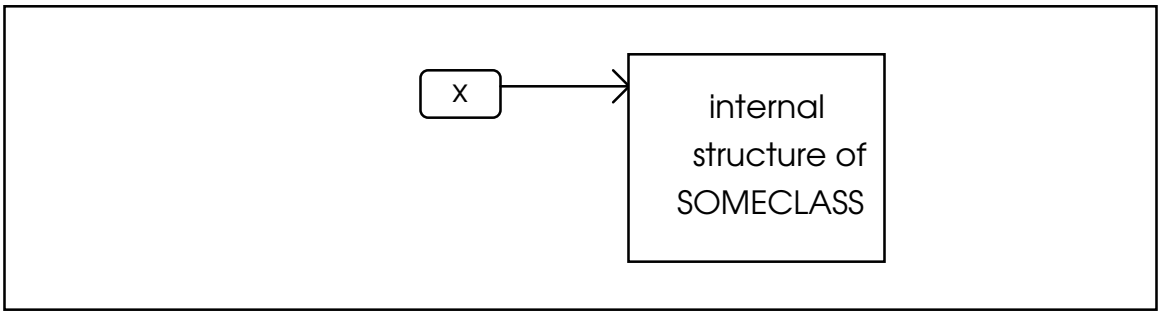
An attempt to reference an object prior to calling a creation (or any other) routine on it will result in an exception. In the above example, *x.some_routine* can only be called after *!!x.make*.

Before *!!x.make* was called, *x* looked like this:



At this point in time, *x* had been declared, but since it was of reference type, the automatic default initialization sets *x* to *Void*. This is similar to NIL or NULL in Modula-2 and C/C++ respectively.

After, the *!!x.make* statement, *x* now looks like:



1.2 Expanded Types

If a class is defined so that all instances of it are expanded, then an object declared of that class type is ready to be used as soon as it is declared. For example, as soon as an object of type INTEGER is declared, it may be used, since INTEGER is an expanded class. Expanded objects (declared of expanded classes) do not require the !! operator to be called prior to usage. It is implicitly called automatically when the object is declared. This gives the flexibility of choice between references to objects and objects themselves (pointers to variables, or just variables).

To declare a class such that all instances of it (objects) are expanded, use the **expanded** keyword before the class declaration:

```
expanded class INTEGER
feature
  ...
end --INTEGER
```

If we want to instantiate an object of the INTEGER class, we simply declare it as follows:

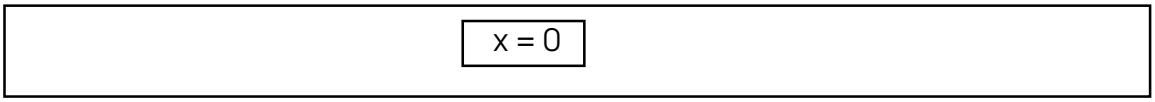
```
x : INTEGER; -- At this point, x has a default value of 0
x := 1;          -- x  now contains the value 1.
```

Part of the automatic initialization causes the object *x* to contain the value 0. In Eiffel, all entities are initialized to a default value. The corresponding default values for the basic types are:

```
INTEGER     =     0
REAL        =     0.0
DOUBLE      =     0.0
CHARACTER   =     Null_char
BOOLEAN     =     False
reference   =     Void
```

The REFERENCE type above is not an actual class type. Rather it denotes a class whose type is not of any of the above listed. That is, if a class is defined and it is not preceded by the **expanded** keyword, all objects of that class type are references. Hence at the instant of declaration, the reference itself is set to Void by default.

Getting back to expanded types: graphically, if we wanted to represent *x* which is an expanded INTEGER, we could do it as such:



The above graphic represents *x* at the point of declaration.

1.2.1 Declaring Expanded Objects

There may come a time when a reference to an object is not needed, and the object itself is more appropriate but not available through the class itself (via **expanded** keyword). In this case, the object itself may be declared as **expanded**. The effect is the same as if the class itself had been declared as expanded as described above. In the case of expanded objects, we may or may not know if the class itself was declared as expanded or not. We may know however, that we wish for the object to be expanded. If this is the case, we may do the following.

```
x : expanded SOMECLASS
```

In this case *x* is of type SOMECLASS, but instead of being a reference that must be initialized (with the !! operator), all memory necessary for SOMECLASS is allocated upon declaration. Each attribute of SOMECLASS is then automatically initialized to its respective default value.

1.3 Features

Features are the data an object may store and operations that may be performed on an object. There are two types of features in Eiffel: attributes and routines. The term *feature* in Eiffel refers to the data and operations collectively. Features allow one to describe what a class has or can do; that is, it supports the 'has-a' or 'can-do' relation. A CAR for instance, will 'have-a' *engine*. A COMPUTER 'has-a' CPU. Also, a CAR 'can' *drive_forward.* A COMPUTER 'can' *compute*.

1.3.1 Attributes - Variables and Constants

An attribute is an object whose declaration is found within a class (aggregation). An attribute may be a referenced class or an expanded class, such as an INTEGER. The following is an example of an attribute **feature** list:

```
feature
  count : INTEGER;
  precision : DOUBLE;
  last_key : CHARACTER;
  icon_selected : BOOLEAN;
```

1.3.1.1 Unique Attributes

Eiffel provides no mechanism for implementing enumerated types. Instead it provides a capability to simulate the effects of enumerated types while still strongly holding on to its object-oriented nature. This capability is manifested in the **unique** keyword. An example is deserved:

```
class FSM
feature
  state_1, state_2, state_3 : INTEGER is unique;
  ...
end
```

The only entity types that may be declared as **unique** are INTEGERs.

Unique INTEGERs are guaranteed to be different from all other INTEGERs declared as **unique** within that class. Here, *state_1, state_2,* and *state_3* are unique because they will each be assigned different INTEGER values. Unique attributes are essentially constants whose value is chosen by the compiler. What this means for the reader is that in essence the functionality of enumerations (assigning INTEGER values to

meaningful identifiers) has been implemented, yet the pure object-oriented structure remains in tact.

Any unique attributes in a class will be assigned numerical values such that each successive declaration is one greater than its predecessor and each entity is also positive. In the case above, if *state_1* was assigned the value 1, then *state_2* and *state_3* would be assigned 2 and 3 respectively. It could be possible that *state_1* was assigned the value 5, in which cast *state_2* and *state_3* would be assigned 6 and 7 respectively.

1.3.1.2 Constants

In Eiffel constants are always declared with an associated type. The only types whose objects can be declared as constants are:

```
INTEGER
REAL
DOUBLE
BOOLEAN
CHARACTER
ARRAY
```

These are the basic types available in most traditional languages. We will now show how to declare an entity as a constant.

```
feature
  Max : INTEGER is 100;
  Pi : REAL is 3.14;
  Better_pi : DOUBLE is 3.14159;
  Default_save_mode : BOOLEAN is False;
  Menu_option : CHARACTER is 'M';
```

1.3.2 Routines - Procedures and Functions

Routines are functions and procedures belonging to a class that provide algorithms utilized by that class. They may facilitate the input and/or output of data to the class or calculate values based on the state of the object. As with attributes, routines may be declared following the **feature** keyword.

For example, the following class converts a Celsius temperature to Fahrenheit. It utilizes a procedure *set_temperature* to set the initial temperature, and a function *convert* to return the converted value.

```
class CONVERTER
feature
  temperature : DOUBLE;
  conversion_factor : DOUBLE is (5/9);
  set_temperature ( c : DOUBLE ) is
  do
    temperature := c;
  end -- set_temperature

  convert : DOUBLE is
  do
    Result := temperature * conversion_factor + 32;
  end -- convert
end  -- converter
```

There are four different types of features in class CONVERTER. The first, *temperature*, is a an instance of type DOUBLE. It is a variable attribute whose value may be changed during program execution.

Second, *conversion_factor* is an instance of class DOUBLE, and is initialized with a value of (5/9). Using the keyword **is** causes *conversion_factor* to be a constant attribute, as described above.

Third, *set_temperature* is a routine that does not return a value. It takes one parameter, *c*, which is of type DOUBLE. The keyword **is** indicates the identifier is about to be assigned meaning, and the **do...end** pair states that this is a routine.

Finally, *convert* is a function that has no parameters and returns a value of type DOUBLE. It assigns the converted *temperature* to the keyword *Result*. *Result* is the pass-back mechanism (an object) for Eiffel. Whatever its value is upon function termination is what is returned to the calling routine.

1.3.2.1 Creation Routines

As noted in an earlier section, objects that are references need to be initialized with a creation routine from that class. A routine is designated as a creation routine by placing it in a list following the **creation** keyword. This routine may then be called on an object along with the (!!) operator to provide initialization. In the example below, *make* is designated as a creation routine.

```
class CONVERTER
creation make;
feature
  temperature : DOUBLE;
  make  is
  do
    temperature := 98.6;
  end -- make
  ...
end
```

In Eiffel, creation routines are not treated any differently than any other routine. That is, it may be called during creation of an object, or it may be called at any time during the life of that object. In the text given above we may call *make* at any time on an object of class CONVERTER if we wanted to set the *temperature* feature to 98.6 degrees. This will not allocate space for a new object of class CONVERTER. Only the (!!) creation operator will do this. We may now declare an instance of class CONVERTER, *widget* for instance, and create it as follows:

```
...
widget : CONVERTER;
!!widget.make;
...
```

By default, since CONVERTER is not defined as an **expanded** class and *widget* is not declared as an **expanded** version of CONVERTER, *widget* is a reference to a CONVERTER object. Prior to using *widget*, it is created (using the !! operator) and initialized with the call to the *make* routine.

1.3.3 Local attributes

Routines and functions may sometimes need local (and temporary) space to help aide in calculations, or for some other reason. Local variables are nothing new to programming languages, nor to Eiffel. Local variables are in fact objects (as are all variables). A local entity may be declared as follows:

```
...
convert : DOUBLE is
local
  loc : DOUBLE
do
  loc := temperature * conversion_factor + 32;
  Result := loc;
end -- convert
...
```

In this case, *loc* is a local variable in the *convert* function. It is only in existence when the function is executing. It does not retain a value in-between calls to *convert*. If instead of local variables, local constants are required, the same syntax and semantics should be applied as those for declaring constants as described in section 1.4.1.3.

Chapter 2:

Inheritance,
Using Objects,
Clients and Suppliers

2.1 Inheriting from Other Classes
2.2 Using an Object - the Dot (.) Operator
2.3 Clients and Suppliers
2.4 Export Status
2.4.1 Multiple Exports

2.1 Inheriting from Other Classes

Being an object-oriented language, Eiffel supports the inheritance mechanism. Specifically, multiple inheritance is fully supported. A class may inherit features from one or more parent classes, declared in the **inherit** clause of a class definition. The following example demonstrates the syntax of Eiffel inheritance:

```
class  CHILD
  inherit
    PARENT_1;
    PARENT_2;

  feature
    ...  -- CHILD's own features
end --CHILD
```

Here, class CHILD inherits from PARENT_1, and PARENT_2. Any features from those two classes become part of class CHILD, in addition to any features defined in CHILD. Problems may arise, though when features with the same name are inherited from PARENT_1 and PARENT_2. The possible ambiguities with inheritance can all be resolved through selection, redefinition, renaming, and undefining. These topics will be covered in detail in chapter 5 which is titled *Inheritance and Feature Adaptation*.

2.2 Using an Object - the Dot (.) Operator

Once an object has been created, the dot (.) operator is used to invoke a routine, or access an attribute. Regardless of the type of the object (reference or expanded), the dot operator is used consistently. In traditional languages that require one to explicitly use the dot operator for expanded classes (or equivalents), and another operator (^. or ->) with reference objects (or equivalents), this may cause a great deal of time to change, if it has been decided to switch from an expanded object to a reference or vice versa. Eiffel's consistent use of the dot operator only is a beneficial feature for both program writing ease and for readability and understandability of concepts rather than implementation details.

We now present the following example to better describe the use of the dot operator:

```
c : CONVERTER;
!!c.make();
c.SetValue(32);
```

Here, the creation operator (!!) along with the call to the creation routine *make*, creates and initializes *c* as an object of type CONVERTER. Then *set_temperature* sets *c*'s *temperature* feature to 32 as described by the actual INTEGER parameter. The dot operator associates *set_temperature* with a particular instance of CONVERTER, here *c*. Any other CONVERTER that may exist remains unaffected.

2.3 Clients and Suppliers

The relationship of inheritance described in section 21 above is known as the 'is-a' relation. For instance, a CAR 'is-a' VEHICLE (CAR inherits from VEHICLE) or a PEN 'is-a' WRITING_IMPLEMENT (PEN inherits from WRITING_IMPLEMENT).

Another relation existing (among others) in the object-oriented paradigm, is the 'has-a' relation, which describes aggregation. For example, a PERSON 'has-a' HEAD, or a DIGITAL_CLOCK 'has-a' DISPLAY. In Eiffel, the 'has-a' relation is implemented simply by stating that in a particular class, one of the features is an attribute.

```
class DIGITAL_CLOCK
feature
  display : DISPLAY;
...
end
```

Here, since DIGITAL_CLOCK is using DISPLAY through its *display* feature, it can be said that DIGITAL_CLOCK is a client of DISPLAY. Conversely, DISPLAY is a supplier of DIGITAL_CLOCK. With this simple concept in mind, we take you to the next section which concerns the export status of features.

2.4 Export Status

There may be times when we do not want to allow just any client to use all of the features of a class. Some clients should be able to use all of the available features, others should be able to use only a select few. In Eiffel, we are able to describe which clients are allowed to access which features (via the dot '.' operator). This is accomplished with a selectable export status mechanism as described below:

```
class ACCOUNT
feature
  balance : INTEGER;
  ...
end --ACCOUNT
```

If we were to let this class stand as-is, we could change the *balance* feature as we wish without penalty. This means that we could set the *balance* to hold a million dollars, and no one would argue the fact (except maybe the bank holding the account!).

We need a way to say that we do not wish for anyone to be able to access *balance* directly. This may be done as follows:

```
class ACCOUNT
feature {NONE}
  balance : INTEGER;
  ...
end --ACCOUNT
```

Now, no class is able to access *balance*, since it is exported to the NONE class only. No objects may be instantiated of type NONE, hence no objects will ever be able to use it. The {NONE} appearing after the **feature** reserved word states that all of the following features are available to objects who are of type NONE.

This seems a little too restrictive for this case. What if the bank needed to access your *balance* in order to make a deposit, or withdrawal. In this case, we can export the *balance* feature to the class BANK as follows:

```
class ACCOUNT
feature {BANK}
  balance : INTEGER;
  ...
end --ACCOUNT
```

Above, only clients of ACCOUNT who are of declared to be of class BANK or who inherit from BANK may directly access *balance*. This restrictive exportation of *balance* enables only those classes who can be 'trusted' to use *balance*. We would no export *balance* to just anyone, but we don't want to enable everyone to have free access to it.

Based on this declaration, the following may be written:

```
class BANK
feature
  presidential_account : ACCOUNT;
  ...
  deposit (amount : INTEGER) is
  do
    presidential_account := presidential_account + amount;
  end;
end
```

The following will show what can *not* be written. Since ROBBERS is a client of ACCOUNT, and the feature *balance* has not been exported to clients of type ROBBERS, the following will not be allowed:

```
class ROBBERS
feature
  hacking_account : ACCOUNT                    ...
  wipe_account is
  do
    hacking_account.balance := 0; --illegal!!
  end
end
```

The ability to selectively declare who is able to access a feature or set of features of a class, is called selective exporting. Eiffel, provides an easy to learn and remember syntax for selectively exporting features or groups features.

2.4.1 Multiple Exports

Instead of exporting to only a single class the *balance* feature of our ACCOUNT, we may wish for a group of classes to be able to access this feature. In this case, we can export *balance* to a group of clients as in:

```
class ACCOUNT
feature {BANK, IRS}
  balance : INTEGER;
  ...
end --ACCOUNT
```

Now, if BANK or if IRS objects are clients of ACCOUNT, they will have access to the *balance* feature. That is:

```
class IRS
feature
  your_account : ACCOUNT;
  check_for_fraud is
  do
    if your_account.balance > 1000000 then
      schedule_audit;
    end;
  end; -- check_for_fraud
end --IRS
```

Since the IRS class has been listed as a client who is able to whom *balance* was exported, the above IRS class is legal. IRS is permitted to change *balance* in *your_account* as it sees fit. So care must be taken when exporting a feature which is an attribute to another class. Trust must be instilled in the client class, here IRS, such that IRS will be not tamper destructively with the *balance* feature. (Yes, trust in the IRS is difficult believe, but sometimes necessary!)

Chapter 3:

Copying, Cloning and Equality

3.0 Introduction
3.1 Copying Objects
3.1.1 Shallow Copy
3.1.2 Deep copy
3.2 Cloning Objects - Deep and Shallow
3.3 Object Equality - Deep and Shallow

3.0 Introduction

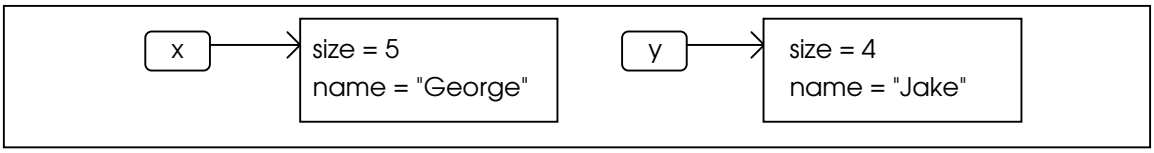
Eiffel provides several features that allow objects to be reproduced, replicated and compared for equality. We will discuss these capabilities in this chapter. Most of the features introduced thus far in previous sections of this paper are not unique to the Eiffel language. However, Eiffel separates itself from other languages in its ability to deeply copy, clone, and compare objects. We will explain in detail why this statement holds true.

3.1 Copying Objects

To copy an object means to replicate the contents of the source object in the target object. In order to be correct, the target object must already be in existence (created). Copying to a target reference that is Void will produce an exception. Eiffel provides two ways to copy an object: deep and shallow copying. Each will be discussed in turn.

3.1.1 Shallow Copy

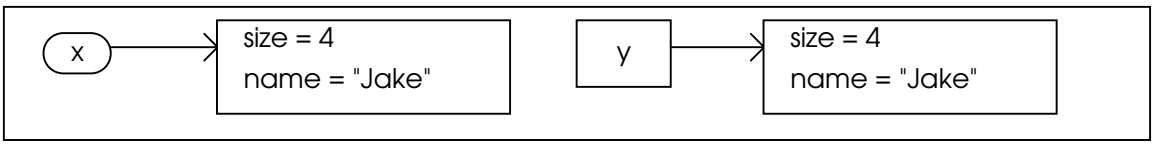
We will discuss the effects of shallow copying with an example. Assume we have two entities which reference objects of class TEDDY_BEAR:



The result of

```
x.copy( y )
```

which is the shallow copy feature available to all objects, would be:



Shallow copy makes no attempt to traverse the structure (other references) of the object. The effects of the copy are isolated to the immediate object being referenced. Shallow copying essentially performs a field-by-field transfer of all features that are attributes. In the above case, *size* and *name*. Note that there are probably routines and

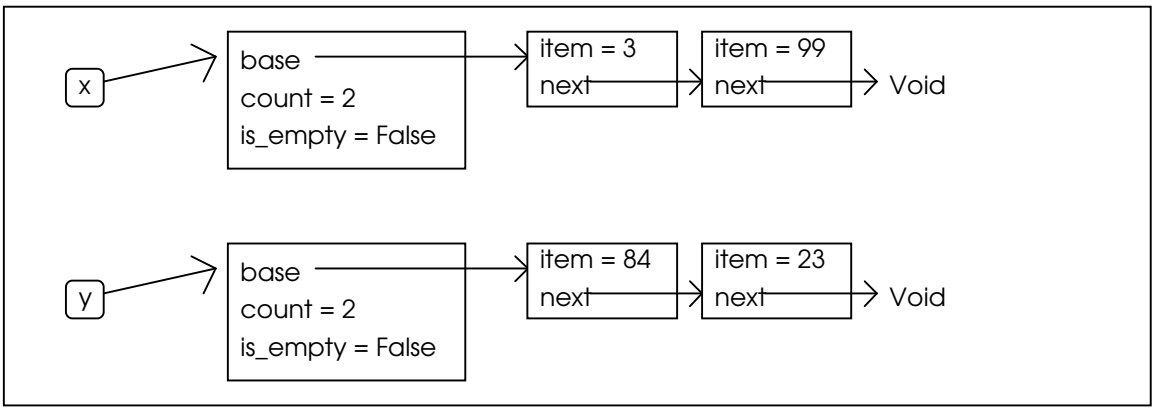
functions available to TEDDY_BEAR that perform various operations. These routines and functions are not copied after a shallow copy (or deep copy for that matter).

3.1.2 Deep copy

Deep copying traverses the entire structure of the object being copied. That is, if we were to deep copy a linked list of 100 elements, each element (all 100 of them!) would be copied onto the destination (which must already have 100 nodes itself).

This feature of Eiffel is where the language begins to separate itself from other traditional and even other object-oriented languages. The ability to traverse an object and copy all of its data to another object is a very useful capability that saves a programmer a tremendous amount of time had the programmer had to write the traversal procedure him or herself.

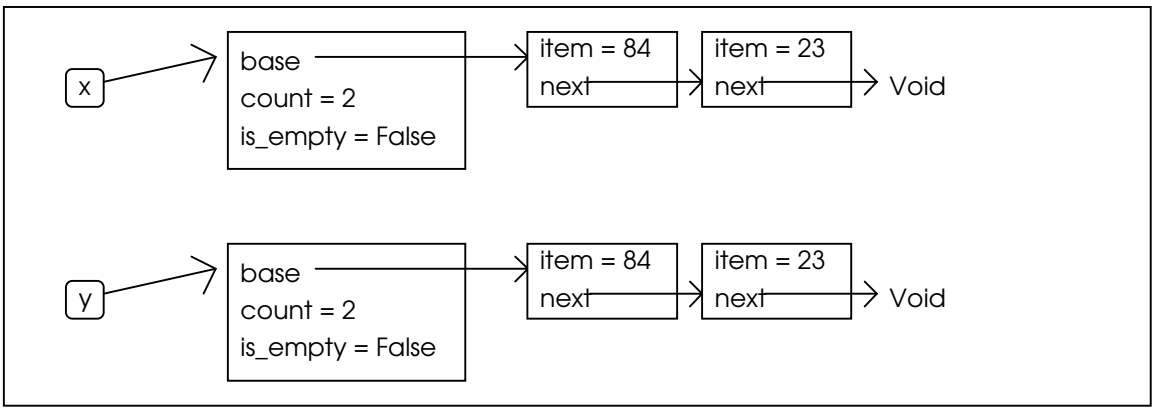
To better illustrate how deep copying works, consider the situation below where *x* and *y* are objects of type LINKED_LIST:



If we now wrote

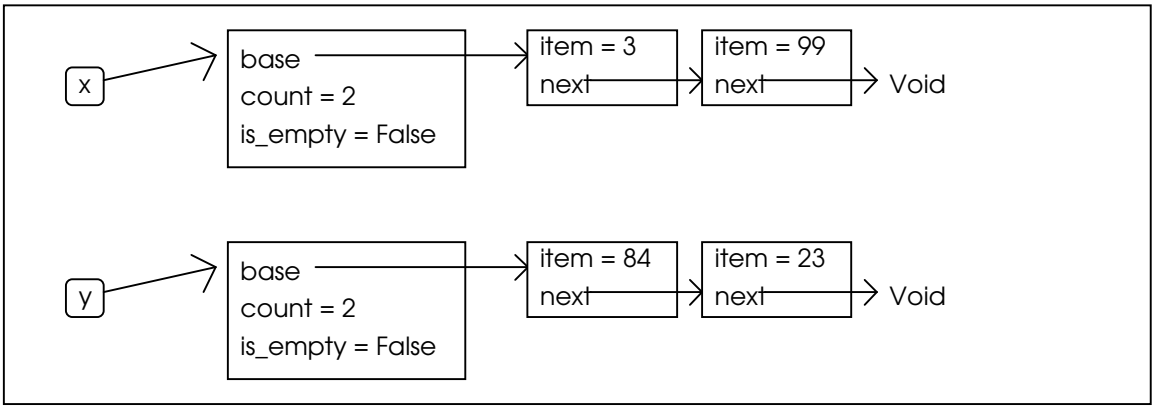
```
x.deep_copy( y )
```

our objects would now look like:

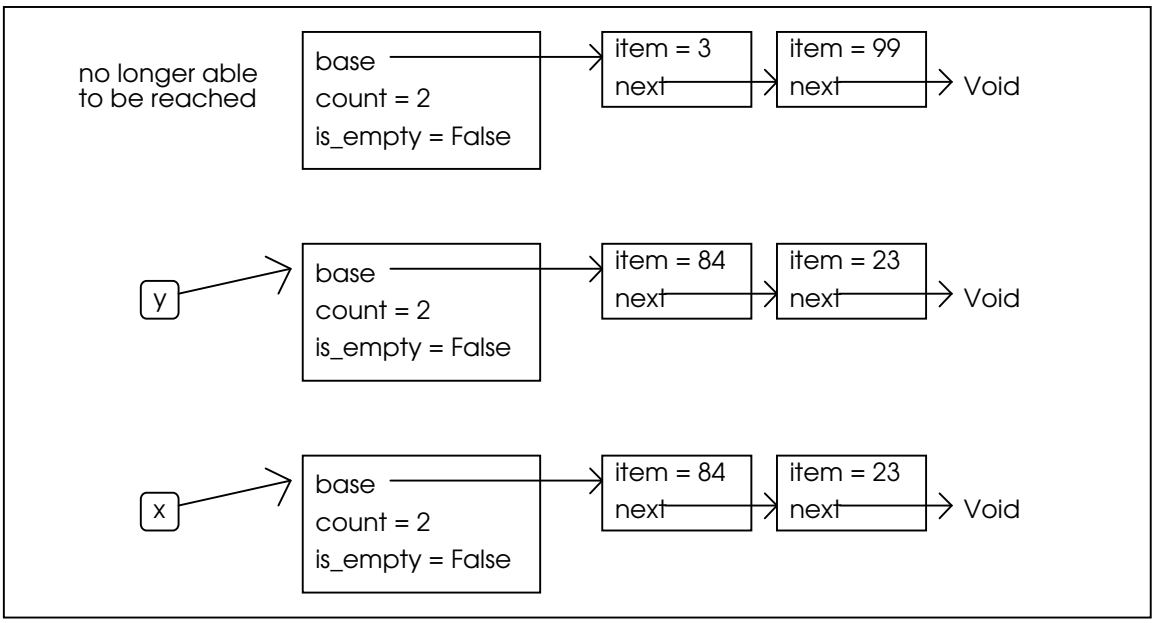


3.2 Cloning Objects - Deep and Shallow

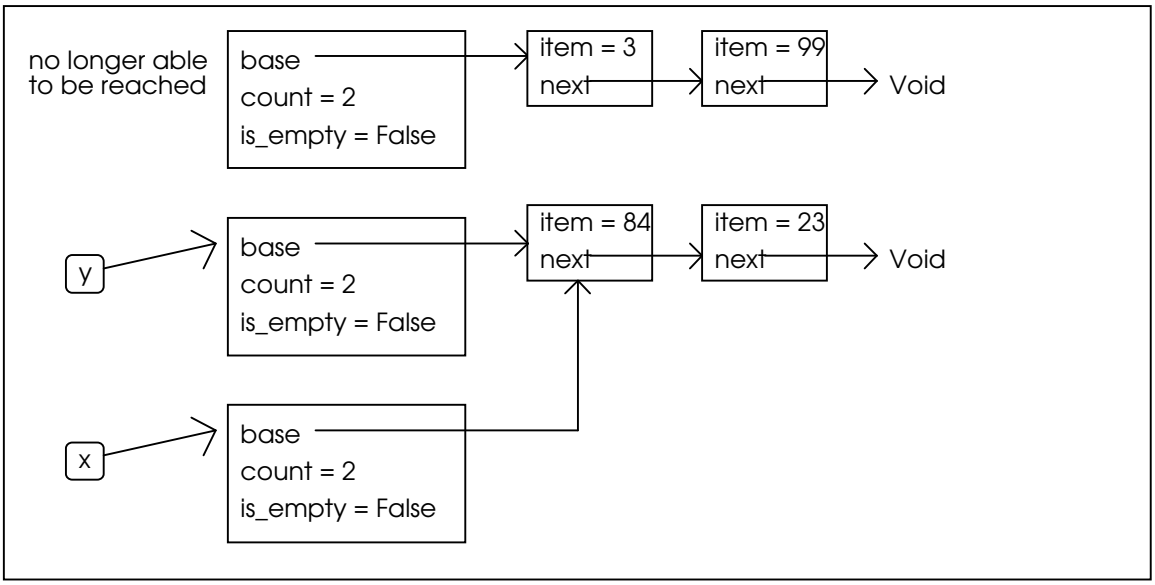
The semantics of cloning objects are similar to that of copying with the exception that instead of copying an object into an already existing object, a new target is created (cloned) from the source. The features available to all classes to perform this are *clone* and *deep_clone* which perform shallow and deep clones respectively. If the target happens to be pointing to an already existing structure, that structure will be lost, and the new clone of the source will be referenced by the target.



Using the example from the previous section concerning copying, if we were to write *x.deep_clone (y)* then the result of this would be:



On the other hand, if we were to write *x.clone (y)* which is a shallow clone, then the results would differ from that of a deep clone. Observe the new *x* and *y* references after a shallow clone:

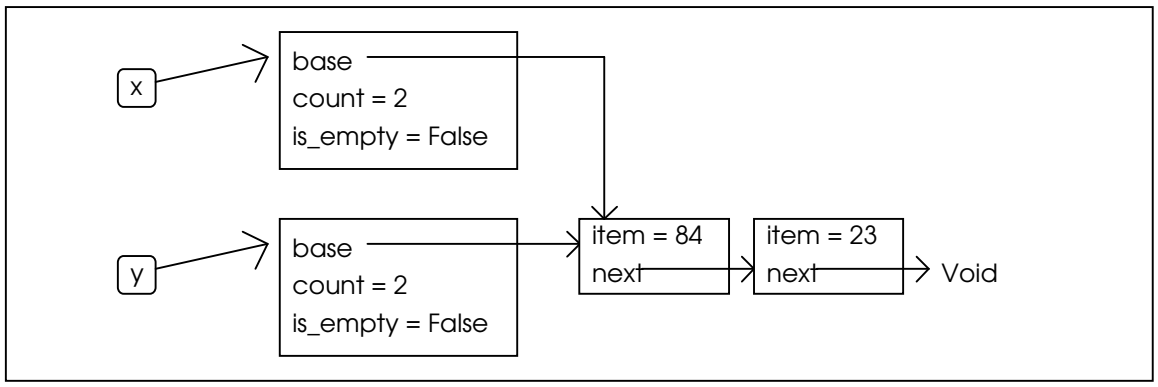


Since the only portion of *y* that was cloned is the immediate object referenced by *y*, the result is shown above. In essence, we have created a new header that references the already existing linked list which *y* is referencing.

3.3 Object Equality - Deep and Shallow

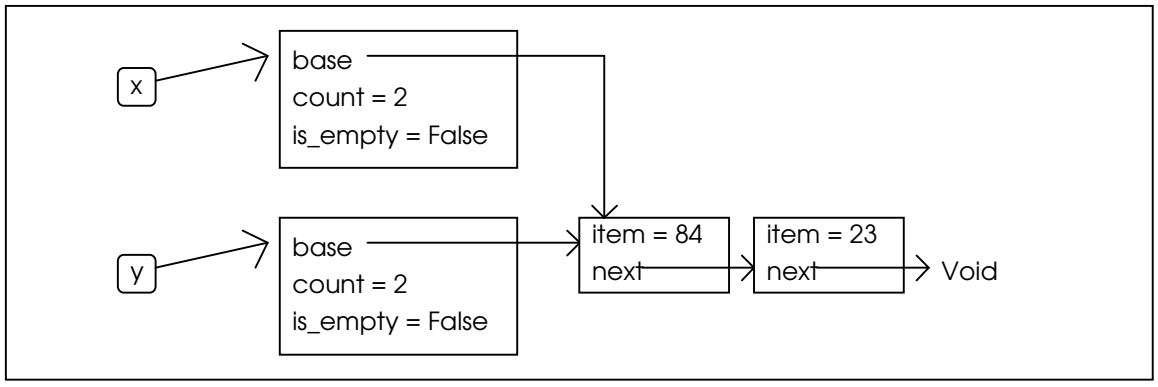
A mechanism for comparing objects for equality is provided. Again, the semantics of shallow and deep equality are comparable to those of the copy or clone feature described above. In the case of equality, both objects being compared (which may be Void) are compared on a field-by-field basis. The features available to all classes for this are *equal* and *deep_equal.*

An example is deserved:



Here, if we were to perform a deep or shallow equal (comparison), the result of both of these operations would be that in fact *x* and *y* are equal (both deep and shallow). They are shallow equal because the immediate objects referenced by *x* and *y* are field-by-field equivalent. That is: *x*'s *base* equals *y*'s *base, x*'s *count* equals *y*'s *count,* and *x*'s *is_empty* equals *y*'s *is_empty*. The shallow equal stops there.

Deep equal goes the distance of traversing each objects in the entire link for field-by-field equality.



This illustration used with the shallow equal section also qualifies for deep equal. That is *x* is said to be deep equal to *y* as well as being shallow equal..

Chapter 4:

Control Flow Constructs - Conditional, Iteration, and Multi-Branch

4.1 Looping
4.2 Conditional Constructs
4.3 Multi-Branch

4.1 Looping

For simplicity, there is only one iterative construct in Eiffel, and that is the **loop** construct. One might ask if it is possible to "get away" with only one looping construct; after all the FOR, WHILE, and REPEAT loops served us well in previous languages. Eiffel's looping construct is generic enough to be able to construct a loop that behaves in the same fashion as each of the above loops. The advantage of only one looping construct is that once it is learned, that is it! There are no variations that allow a programmer to dangerously change the nature of a loop. This promotes a simple and easy to learn construct that gives all of the benefits of each of the above loops. A simple example of the **loop** construct is as follows:

```
from
  --initialization
until
  done -- BOOLEAN condition
loop
  -- executable statements
end
```

The **from** clause is mandatory, and it specifies initializing statements. In this example, no initialization is performed.

A second mandatory clause is **until;** it causes the loop to stop executing when it's expression evaluates to *True*. More than one expression may be supplied, in which case each one is separated with a semicolon ";"

The **loop...end** is the body of the loop to be iterated. All actions that are to be executed during the iteration are placed in here.

It would be best if we illustrated how to mimic the behavior of the FOR, WHILE and REPEAT loops using Eiffel's **loop** construct. We will first start with the FOR loop which is characterized by a fixed, pre-determined number of iterations. We may construct a loop that acts like a for loop as follows:

```
from
  i := 1
until
  i = 10
loop
  io.put_int (i);
  io.new_line;
  i := i + 1;
end
```

The way we achieve a FOR loop mimic is by manually incrementing the counter variable *i* so that the effect of automatic incrementation is constructed. It is not much more effort to include this manual increment statement while preserving the consistency and generality of the **loop** construct.

To program a WHILE loop:

```
from
  node := first;
until
  node = Void;
loop
  io.put_string (node.text);
  node := node.next;
end;
```

This **loop** is similar to saying that WHILE *node* is not *Void*... execute the loop body. The effect of a traditional WHILE loop- testing for loop termination prior to the initial loop execution and all subsequent iterations of the loop body, is done.

We may perform similar actions when a REPEAT loop is desired. There is a slight difference, though and that is we must ensure that the **until** clause will evaluate to *False* on the first iteration so that in effect the **loop** will be guaranteed to execute at least once. We can achieve this as follows:

```
from
  done := False
until
  done and pages = 10;
loop
  done := True;
  printer.output_next;
  pages := pages + 1;
end
```

Since the *done* expression (a BOOLEAN variable) initially evaluates to *False*, the **until** clause will initially evaluate to *False* hence guaranteeing at least one execution of the **loop** body.

4.2 Conditional Constructs

Like other high-level languages, Eiffel has the **if...elseif...end** construct. It is fully bracketed (each **if** must have a closing **end**). An example of this is as follows:

```
...
if x > 10 then
   ... statements ...
elseif x > 5 then
   ... elsif statements ...
elseif x > 0 then
   ... more elsif statements ...
else
   ... else statements ...
end
```

Since the functionality of this (and most other) **if** statements is fairly trivial, no other mention will be made hereafter.

4.3 Multi-Branch

Instead of coding extensive **if...elseif** statements, Eiffel supports a multi-branch construct, **inspect..when**. This construct mimicks the CASE or SWITCH statement of most other languages. Its usefulness is comparable to that of the traditional constructs mentioned above.

```
inspect  input_character;
when 'y' then
   ... statements
when 'n' then
   ... statements
else
   ... if no match for input_character is found
end
```

Here, the appropriate statements are executed depending on the value of *input_character*. In this case, if *input_character* is neither 'y' nor 'n', then the statements in the **else** clause are executed.

The following example shows how a multi-branch instruction can be combined with **unique** attributes to imitate a CASE statement that might be based upon the value of an enumerated variable:

```
...
State : INTEGER;
```

```
State_1, State_2,  State_3 : INTEGER is unique;
...
inspect State
when State_1 then
  some_action;
when State_2 then
  some_other_action
when State_3 then
  another_action;
else
  no_action;
end;
```

This example analyzes *state*, and takes the appropriate action depending on its value. If *state* is some other value, then the **else** clause is executed. Again, the effects of the **inspect** construct are trivial and will not be discussed any further.

Chapter 5:

Inheritance and Feature Adaptation

5.0 Introduction
5.1 Selecting
5.2 Renaming
5.3 Redefining
    5.3.1 Feature Signature
5.4 Changing the Export Status of Inherited Features
5.5 Undefinition
    5.5.1 Semantic Constraints on Undefinition
        5.5.1.1 Frozen Features

5.0 Introduction

Perhaps the most potent contribution to software reusability and adaptability is the notion of inheritance. Using this powerful facility, software construction becomes a simple task of adapting previously written, and more importantly, previously tested code. It is almost paradoxical, though in that this important feature of the object paradigm is almost never fully recognized in most programming languages that claim to be object oriented. Some may offer a single inheritance mechanism, others offer multiple inheritance, but do not provide facilities to deal effectively with such ambiguities as name collisions, and repeated inheritance.

Eiffel, on the other hand, is very unique in these respects. The language provides mechanisms to enable all of the benefits of multiple inheritance to be used to their fullest consequences. Eiffel allows a descendant class to rename, redefine, undefine, select, effect (in the case of deferred features), and change the export status of virtually all inherited features. Of course, there are semantic restrictions placed on the new class when attempting to adapt an inherited feature in an ambiguous, or perhaps unsafe manner.

5.1 Selecting

To start our journey into the vast combinations of feature adaptations, we will take a look at the selection mechanism. For a given class C which inherits from at least two classes, say A and B, there exists a possibility that A and B will both have a feature named *f* for instance. If C inherits from A and B, we have what is called a name collision or clash. The problem of this ambiguity may be more prominently displayed in this example:

```
class A
feature
  f : INTEGER;
end

class B
feature
  f:: REAL;
end

class C
  inherit A;
          B; -- Illegal!
  end
end
```

The question is then, how to resolve the two inherited features with the same name *f*, from A and B. The select clause offers one solution. By modifying the text of C:

```
class C
  inherit
    A select f;
    B
  end;
feature
  ...
end
```

we explicitly state that we wish to allow *f* to be inherited from A and not from B, thus resolving the clash. Feature *f* from B is not inherited in class C, although if the select clause were to be placed in B's inherit clause as in:

```
class C
inherit
  A;
  B select f;
end;
feature
...
end
```

the opposite would hold true. That is, *f* would be inherited from B, and not from A.

Without a select clause, one could not tell whether the run-time environment would dynamically bind *f* in C to A's or B's version. If the compiler does not permit this ambiguity, then it must report to the programmer that such an inheritance situation is a violation of the language. Clearly, there may very well be a reason to inherit in this manner; a simple naming duplication should not prevent one from attempting inheritance.

5.2 Renaming

In some contexts, it may be desirable for a descendant class to inherit two features identically named *f* from two different classes A and B. In this case, selecting just one *f* from either A or B will not sufficiently enable one to acquire both versions of *f*. Recall that selection enables only one version of *f* to be inherited from any set of parents.

However, we can rename one version of *f* to *g* for example, thusly resolving a name clash, if any, and enabling two versions of *f* to be acquired, one of which is under a different name. The following revision of class C describes this situation:

```
class C
inherit
  A
    rename f as g;
  B;
end;
feature
...
end
```

In this situation, A's version of *f* has been renamed to *g*. This has dual purpose. First it resolves the name clash of the two versions of *f* from A and B. Second it allows both versions of *f* to be inherited in C, A's version under a different name, giving greater control to the programmer concerning how features should be adapted according to how he/she sees fit.

One can also use this facility in the absence of a name collision. For instance, if a feature is not descriptive enough to match its new surroundings, it may be renamed to reflect its new role in the system.

```
class EMPLOYEE
inherit
  PERSON
    rename occupation as company_division
  end;
feature
...
end
```

In the above context, since we know that an EMPLOYEE's occupation is working for the company of which EMPLOYEE refers, it is possible to rename *occupation* as *company_division* and still keep all of the semantics intact.

This poses one to question the validity of other features inherited from PERSON which reference *occupation*. That is, if a feature from PERSON depends on the *occupation* field, will it still be a valid feature since *occupation* doesn't exist anymore? Consider:

```
class A
feature
  x, y : INTEGER;
  set_x (new : INTEGER) is
    do x := new end;
end
```

```
class C
inherit
  A
    rename x as x1
  end
end -- C
```

Now, if we create an instance of class C, and attempt to call the *set_x* routine, is this legal? The answer is yes, this is perfectly legal. The fact that *x* has been renamed to *x1* does not affect the routine *set_x* (although it should be stated that *set_x* should be itself renamed to *set_x1* for consistency on the programmer's part. The compiler does not require you to rename all references to a renamed feature). Although, if another routine say *double_x1* which doubles the value of *x1* is to be written in C's context, it must refer to *x1* and not *x*. The effect of renaming *x* to *x1* also makes the identifier *x* available to C. That is, C itself may declare a feature *x* that is totally different than the original *x* from A.

We have seen thus far the two conventions of selecting and renaming features to adapt them to their new environment. Eiffel provides another mechanism to adapt a feature to its new environment.

5.3 Redefining

A situation commonly encountered in software development is when a feature does not necessarily conflict with another feature's name, but it might be suitable just to enhance or specialize a feature to reflect an algorithmic improvement or other benefit of such nature. This is where redefinition enters the scene. Redefining an inherited feature allows one to properly adapt it to its new environment. This form of software Darwinism is one of the most attractive capabilities of the object paradigm.

Before we give actual examples of redefinition, it should be said, though that there is a considerable constraint placed on feature redefinition that the selecting and renaming clauses did not impose. This constraint is that the new signature of a feature *f* must conform to its original signature. It now seems appropriate to discuss the exact meaning of the signature of a feature.

5.3.1 Feature Signature

The signature of a feature *f* is the pair (A,R) where A is the sequence of argument types of a routine or function (empty for entities); R (empty for routines) is the return type for a function, or just the entity type for entities. The following examples make this more clear:

Feature	*Signature*
variable_attribue (entity)	
x : INTEGER	(<>, <INTEGER>)
y : REAL	(<>, <REAL>)
function_without_arguments	
day_of_year : INTEGER	(<>, <INTEGER>)
function_with_arguments	
func(x : INTEGER; r : REAL) : REAL	(<INTEGER, REAL>,<REAL>)
func(a : SOME_CLASS; x : INTEGER)	(<CLASS,INTEGER>,CLASS)
procedure_with_arguments	
proc(a : CLASS; r : REAL; x : INTEGER)	(<CLASS,REAL,INTEGER>,<>)
proc(x, y : INTEGER)	(<INTEGER, INTEGER>, <>)

Simply put, the signature of a feature is the sequence of all types of arguments, if any, and the return type (in the case of a function); or just the attribute type in the case of an entity.

What does this have to do with redefinition? As mentioned above, there is a constraint placed on a class that wishes to redefine an inherited feature. That is, the new redefined feature's signature must conform to the signature of the feature which is being redefined. An example is deserved:

```
class A
feature
  set_all( x, y : INTEGER; r : REAL ) is
    do ... end
end
class B inherit
  A
    redefine set_all
  end;
feature
  set_all( i, j : INTEGER; s : REAL ) is
    do ... end;
end
```

Here we have the case of class B redefining *set_all* such that its routine body is algorithmically different from that of A's version of *set_all*. Notice that the signature of *set_all* defined in B conforms to that of the signature of *set_all* in A:

Feature	*Signature*
set_all(x, y : INTEGER; r : REAL)	(<INTEGER,INTEGER,REAL>,<>)
set_all(i, j : INTEGER; s : REAL)	(<INTEGER,INTEGER,REAL>,<>)

What is the difference between conforming signatures and matching signatures? A signature S is said to conform to a signature P if the following holds true:

Signature Conformance

1. The number of types in the argument sequence of both signatures match exactly.
2. Each type in S's argument sequence is a descendant of the corresponding type in P's sequence. Remember that a type is also a descendant of itself so that corresponding types in the sequences may be identical.

What this means is that the number of types in both the argument and return type sequences of P and S must be exactly the same. The second rule states that a redefined feature is not required to use the exact same types in its arguments and return types; it may use a descendant of any or all of the types, as it sees fit.

Getting back to our example from above, observe the following:

```
class A
feature
  set_all( x, y : INTEGER; r : REAL ) is
    do ... end
end --A

class NEW_INTEGER
inherit
  INTEGER end;  - You CAN inherit from INTEGERs!!
feature
...
end --NEW_INTEGER

class B
inherit
  A
    redefine set_all
  end;
feature
  set_all( i, j : NEW_INTEGER; s : REAL ) is
    do ... end
end --B
```

Is this legal? Absolutely! Since the signature of *set_all* in B does indeed conform to that of *set_all* in A, this is legal. NEW_INTEGER was defined as a descendant of INTEGER, making INTEGER an heir of NEW_INTEGER. The new signatures

Feature	***Signature***
set_all(x,y:INTEGER;r:REAL)	(<INTEGER,INTEGER,REAL>,<>)
set_all(i,j:NEW_INTEGER; s:REAL)	(<NEW_INTEGER,NEW_INTEGER,REAL>,<>)

do not match, but they **do** conform. That is, B's *set_all* conforms to A's *set_all*. It is obviously impossible for the opposite to be true. A's *set_all* signature does not conform to that of B's *set_all* signature.

5.4 Changing the Export Status of Inherited Features

Recall from section 2.4 that we could explicitly export a particular feature (or features) to a certain class or classes. For instance:

```
class VEHICLE
feature {DRIVER}
  identification : INTEGER;
end
```

declares that the feature identification is available only to clients who are either descendants of class DRIVER, or are themselves of class DRIVER. The question then arises as to what happens to this rule when we do the following:

```
class LAND_VEHICLE
inherit
  VEHICLE
end
```

Does identification lose its restrictive export status for a more general one (i.e. available to ANY class)?

Since we did not explicitly redeclare the export status of identification when we inherited from VEHICLE, the status of this feature remains the same.

However, we could have explicitly changed the export status of this, and any other inherited features for that matter, with the export clause:

```
class LAND_VEHICLE
inherit
  VEHICLE
    export {DMV_PEOPLE} identification
  end;
end;
```

This states that any class wishing to use LAND_VEHICLE as a client, can only access identification directly if it is an heir of DMV_PEOPLE (of course, DMV_PEOPLE classes themselves may access identification). There is no restriction placed upon descendants of a class who wish to redeclare the export status of a feature. That is, referring to LAND_VEHICLE above, there is no requirement to newly export identification to a class that is related to DRIVER. The previous export status of a feature imposes no restrictions upon the descendant classes wishing to modify it.

We could also have declared a group of classes that we wish to export identification to, instead of just DMV_PEOPLE:

```
class LAND_VEHICLE
inherit
  VEHICLE
    export {DMV_PEOPLE, MECHANIC, FBI}
        identification
  end;
end
```

In this case, when DMV_PEOPLE, MECHANIC, or FBI are clients of LAND_VEHICLE, they have unrestricted access to *identification*. The initial export status of a feature imposes no limitations when redeclaring its export status. We could have just as easily exported *identification* to NONE, or ANY, or any combination of classes in-between.

For syntactical ease, Eiffel provides a mechanism (a keyword) that allows one to export all features of an inherited class to one class, or group of classes. When using the keyword **all**, you may declare that all features from a particular class are now made available to a certain client, or group of clients:

```
class FEATURES_A_PLENTY
inherit
  FEATURES_GALORE
    export {LUCKY_CLASS} all
  end
feature {NONE}
  private_data : INTEGER;
end
```

Assume that FEATURES_A_PLENTY is inheriting many features from FEATURES_GALORE. The export status of every feature from FEATURES_GALORE is now made exclusively available to LUCKY_CLASS and all of its descendants. The feature *private_data* has been declared in FEATURES_A_PLENTY, and is not subject to the export clause of FEATURES_GALORE. No clients are able to directly access *private_data* since it is exported to NONE.

5.5 Undefinition

A rather interesting part of Eiffel's feature adaptation capabilities is the ability to undefine a feature. That is, to effectively ignore its existence in an inherit clause. This poses some semantic questions that will be explored later. For now, consider:

```
class NEW_CLASS
inherit
  OLD_CLASS
    undefine redundant_routine;
  end
end
```

The redundant_routine which would have been inherited from OLD_CLASS no longer exists in NEW_CLASS. By undefining it, we also free its identifier; that is, we may declare a feature redundant_routine in NEW_CLASS if desired.

5.5.1 Semantic Constraints on Undefinition

Undefining a feature may interfere (unintentionally) with a dependency between the undefined feature, and another feature. Observe:

```
class DEPENDENT
feature
  small_routine( n : INTEGER ) is
    do ... end;
  big_routine( arguments... ) is
    do
      ...
      small_routine( x ); --call small_routine
          ...
    end;
end
```

This class appears to be perfectly normal in its organization, and may be a class that is commonly used. If we try to undefine *small_routine* in a new descendant class as such:

```
class NEW_DEPENDENT
inherit
  DEPENDENT
    undefine small_routine end
end
```

we would be halted in our footsteps by our compiler. Clearly the fact that *big_routine* requiring *small_routine* to execute properly is enough to prevent us from undefining *small_routine*.

There are two more constraints placed upon undefinition clauses. The first is straight-forward: you may not undefine a feature that is an attribute. That is, only routines and functions may be undefined.

5.5.1.1 Frozen Features

The next constraint concerns **frozen** features which may only be routines or functions. Any frozen feature of an heir may not be undefined, redefined or renamed in a descendant:

```
class THERMOMETER
feature
  frozen decrease_temperature is
do ... end;
end

class THERMISTOR
inherit
  THERMOMETER
    undefine decrease_temperature end; --Illegal!
end
```

THERMISTOR's attempt to undefine *decrease_temperature* is not allowed. This semantically makes sense because any feature that is declared frozen should not be able to be undefined, redefined, or renamed.

The benefit of undefinition becomes more clear in an ambiguous multiple inheritance scenario:

```
class A
feature
  f : INTEGER;
end;

class B feature
  f( arg1, arg2 : REAL ) is
    do ... end;
end

class C inherit
  A;
  B end;  -- invalid!!
end;
```

Recall from section 4.1 of this chapter that selection enabled us to select which version of *f*, either from A or B we wish to inherit in C. If we modify C as follows:

```
class C
inherit
  A;
  B
    undefine f end;
end;
```

it now becomes possible to inherit *f* from A, and not from B. How is this different from selecting B's version of *f* with a select clause?

```
class C
inherit
  A
    select f end;
  B;
end;
```

Chapter 6:

Genericity

Section Outline

6.1 Unconstrained Genericity

6.2 Constrained Genericity

6.0 Overview

Genericity is usually applied in the context of container classes. A data structure such as a STACK, QUEUE or TREE can be categorized as a container class in that their main function is to store other objects. In conventional programming, it has been notoriously difficult to write a stack ADT, and be able to apply its abstract operations, push, pop, etc., to a wide variety of data structures. Languages with weak typing rules, such as C allow low-level work arounds such as type casts to circumvent their own limitations. A second approach taken is to work on an even lower level using bytes (or words) to transfer a data structure into or out of a container piece by piece. The end result of these concoctions is a working, but extremely unstable, and unadaptable container structure.

6.1 Unconstrained Genericity

A generic class is a class that accepts type paramaterization. For example, a STACK class may be declared to accept only INTEGERs, VEHICLES, or DMV_PEOPLE. If this is the case, it is said to be generically constrained. This topic is covered in the next section. For now, we will focus on unconstrained genericity, a simpler and more general form of genericity.

To declare a class as generic, we present this simple example which declares a stack that may be declared to accept objects of any type:

```
class STACK [T]
feature
...
end
```

The only new syntax introduced in this example is the generic parameter [T] after the class STACK declaration. The square brackets simply introduce STACK as **generic**, and T is the **formal generic parameter** of each instance of STACK.

Since the STACK was declared to be paramaterizable by a class, we must pass as an actual parameter, a class type, rather than an object entity when instantiating this stack. That is, we must pass the name of a class when declaring a STACK, instead of the name of an object entity. This parameter, TOKEN and SEM below, is called the **actual generic parameter**. Observe:

```
class LUNCH_COUNTER
feature
  tray_stack : STACK[ TRAY ];
  napkin_dispenser : STACK[ NAPKIN ]
  ...
end;
```

We have now declared two STACKs. The first, *tray_stack*, accepts objects of type TRAY, the second, *napkin_dispenser*, accepts napkins of type NAPKIN. The next obvious question, then is how to use this information when we are writing the stack's internal operations like push and pop. Observe:

```
class STACK [T]
feature
  push( element : T ) is
  do
    storage( pos ) := element;
    pos := pos + 1;
  end;
end
```

Viola! We are able to use T anywhere in the context of STACK's body for our purposes.

This implementation is not realistic though, in that we do not have an entity called *storage*, but it could be implemented in such a manner if chosen. Similarly, *pop, top*, and the rest of a STACK's operations are implemented in the same manner. What would an actual use of an unconstrained class look like?

```
class LUNCH_COUNTER
feature
  tray_stack : STACK[ TRAY ];
  napkin_dispenser : STACK[ NAPKIN ]
  ...
  init is
    do
      ...
      napkin_dispenser.pop(
    end;
end;
```

We now have the capability to create stacks that will accept objects of any given type. This genericity is called unconstrained because of its unrestrictive nature. It should be noted though, that if we declared a new class type:

```
class FANCY_TRAY
inherit
  TRAY end;
end;
```

then all entities declared as TRAY would have every right to take part in the *tray_stack* activities. That is, the following is legal:

```
class LUNCH_COUNTER
feature
  tray_stack : STACK[ TRAY ]
  ...
  init is
    do
      ...
      tray_stack.push( new_tray )
       ...
    end;
end
```

Do not think that because STACK is unconstrained, that we are allowed to arbitrarily push objects of any type on to *tray_stack*. The fact is that we are able to push any object declared as type TRAY, or any of its descendants onto *tray_stack*. Remember, Eiffel's type system is based entirely on classes. The fact that the STACK class is generically unconstrained allows us only to declare a single STACK that holds any single type and all its descendants, not a stack that holds all types simultaneously.

Is this dangerous? Not at all. As stated before, older programming languages would mimic this action by type-casting any foreign data structure that was to be placed onto the stack. Eiffel has an extremely strong, and tight typing system. All objects that are placed onto this stack, or any other generic structure you create, retain their identity for the duration of the system's life. If we were to place TRAYs and NEW_TRAYs onto *tray_stack*:

```
tray_stack : STACK [TRAY]
t : TRAY;
new_t : NEW_TRAY;
...
tray_stack.push( t );
tray_stack.push( new_t );
```

then the objects *t* and *new_t*, still retain their identities. That is, *t* is still a TRAY, and *new_t* is still a NEW_TRAY.

Eiffel provides no facilities to allow a programmer to type cast an object to force a potentially dangerous conformance. These dangerous practices are no longer needed as Eiffel provides this mechanism to safely harness this benefit of the object paradigm's full potential.

6.1.1 Multiple Paramaterization

If one could find a use for, one could also declare a class that is multiply parameterized as in:

```
class MULT_PARAM [T, P]
feature
...
end;
```

This simply requires a declaration of MULT_PARAM to provide two type parameters, such as:

```
some_entity : MULT_PARAM [VEHICLE, APPLE]
```

A potential use of this ability might be with a container structure that holds several types of objects. A home entertainment cabinet for instance, holds VCR tapes, cassettes, CD's, Laser Discs, etc.

6.2 Constrained Genericity

Up until now, we have witnessed the ability to declare a generic structure that can be parameterized by another class of any type. A simpler as well as necessary version of generic class are constrained generic classes. A constrained generic class declares that a class conforming to a particular type is the only candidate for instantiation. For example:

```
class BIN_TREE [T -> COMPARABLE]
...
end
```

this class declares that in order for a BIN_TREE to be declared, the actual generic parameter provided must conform to the COMPARABLE class. The new syntax of the -> after the formal parameter T states that BIN_TREE is now generically constrained to accept entities who conform to the COMPARABLE type.

As the reader probably knows, binary trees are structures characterized by quick access to its elements. But, the reason why binary trees have this ability is because its elements are in a sorted order. This implies that its elements are arranged in such a way that given any two elements from the tree, we can compare them (or a characteristic of them) and determine which one is ordered before or after the other. We would not be able to put a objects such as ORANGEs on a binary tree, unless of course there was some characteristic that would enable us to determine if ORANGE A is greater or less than ORANGE B.

Constrained genericity serves the purpose, as displayed above, of guaranteeing that an actual generic parameter has certain characteristics that are required for the semantics of the generic class to remain valid. If we were to declare a class such as:

```
class ELEMENTS
inherit
  COMPARABLE end;
end;
```

then we would be able to also declare:

```
class AVL
feature
  tree : BIN_TREE [ELEMENTS] end;
end
```

This creates an entity called tree that accepts ELEMENTS as elements. The specific features of class COMPARABLE that are required for a BIN_TREE to work properly are the actual comparison routines which BIN_TREE utilizes when determining where elements are to be placed in the tree. Without these, BIN_TREE would not be able to execute properly.

Using the genericity mechanism, it is now possible to create completely (or partially) generic data structures with total safety, reliability and abstraction.

Advanced Topics
Chapter 7:

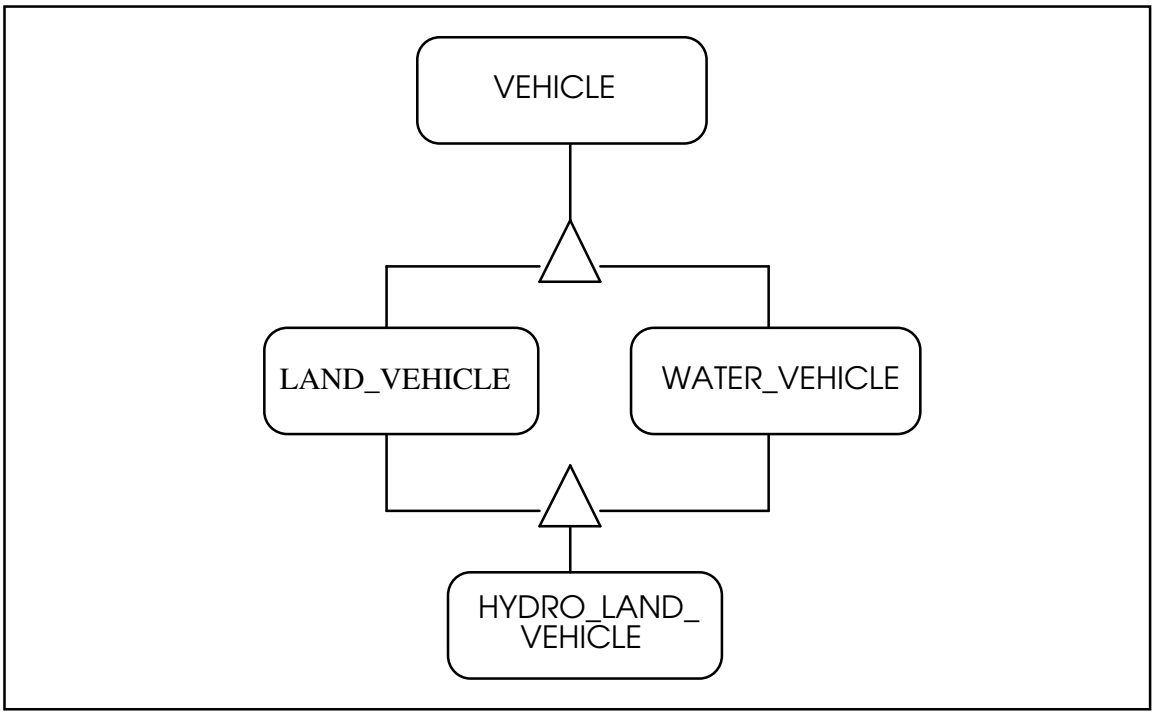
Using Eiffel: Examples and Tutorials

7.0 Overview
7.1 Class VEHICLE
7.2 Contract Programming
    7.2.1 Preconditions
    7.2.2 Post conditions
        7.2.2.1 Old Expressions
    7.2.3 Invariants
        7.2.3.1 Looping Revisited - Variants and Invariants
    7.2.4 A Note on Efficiency
7.3 The New VEHICLE class
7.4 Exceptions
    7.4.1 Handling Exceptions - Rescue clauses
    7.4.2 Retry Commands
7.5 LAND_VEHICLE and WATER_VEHICLE
7.6 Repeated Inheritance
7.7 HYDRO_LAND_VEHICLE

7.0 Overview

The preceding chapters have shown how to use Eiffel in a select number of specific cases. This chapter will take us through select stages of the software life cycle of a small group of related classes. A class called VEHICLE and some of its descendants will be the focus of our attention.

We will apply the principles we have learned in the previous chapters to create a small hierarchy of classes. We will also cover materials not mentioned in previous chapters as we are building our system. The hierarchy for the classes to be described is presented below:



7.1 Class VEHICLE

In this section we will informally describe the requirements for a class called VEHICLE. The specification is given below and is only meant as a starting point for the reader. It is by no means a technical specification.

- The class must provide a facility that enables each instance to be uniquely identifiable.
- Attributes:
  - Color
  - Number of passengers
- Routines:

- Routines to set and retrieve the above attributes
- A constructor routine is also required. It must accept a valid number of passengers that the vehicle may hold as argument.

Based on the above information, several implementations of this VEHICLE class may manifest from different individuals' interpretations. Here is our implementation:

```
class VEHICLE

creation start;

feature {NONE}
  color, num_passengers : INTEGER
    -- "private" data exported to NONE

feature
  Red, Blue, Green : INTEGER is unique;
        -- constants available to all clients

  start (new_num_passengers : INTEGER) is
  require
    new_num_passengers >= 1;
  do
    color := Red; --default
    num_passengers := new_num_passengers; --default;
    io.put_string ( "Hi!. I'm a new VEHICLE" );
    io.new_line;
  ensure
    color = Red;
    num_passengers = new_num_passengers;
  end; --start

  set_color( new_color : INTEGER ) is
  do
   color := new_color;
  end --set_color

  set_horse_power( new_power : INTEGER ) is
  do
    horse_power := new_power;
  end --set_horse_power

  get_color : INTEGER is
  do
    Result := color;
```

```
        end --get_color

        get_horse_power : INTEGER is
        do
          Result := horse_power;
        end --get_horse_power

        get_id : INTEGER is
        do
          Result := object_id;
        end --get_id

      end --VEHICLE
```

The first question that might pop into your head probably concerns the *object_id* attribute in *get_id*. Where did it come from? The answer is it was inherited from class ANY. Recall that all user-defined classes implicitly inherit from ANY. This field is designed specifically for the purpose of identifying objects, even if of the same class, uniquely. This attribute is indeed unique for each object in the system at any given time.

We have now fulfilled the specifications of the VEHICLE class given earlier. We have all of the required attributes, as well as routines to change these attributes and examine them.

7.2 Contract Programming

While these routines and attributes do indeed fulfill the requirements of the specification given, the class is not entirely safe. That is, a client at this point in time is allowed to pass non-sensible arguments to all of the routines without consequence from our class. We need restrictions placed upon this class to ensure that any clients using our class will not be allowed to tamper with it, yet be able to fully utilize all of its features.

Eiffel embodies a type of programming; contract programming, as it is called, provides a premise that allows clients and suppliers to fully recognize and understand certain requirements that they must meet, and guarantees that they are entitled to. These conditions are enforced through assertions, or conditions that a class must meet if it wants to interact with other classes. These rules of behavior manifest in preconditions, post conditions, and invariants.

7.2.1 Preconditions

A precondition is a requirement placed on the client of a class. The precondition itself is specified by the supplying class, and must be met by any clients. An example might be that the pop operation for a stack will require a non-empty stack:

```
class STACK [T]
feature
  is_empty : BOOLEAN;
  pop : T is
    require
      non_empty_stack : not is_empty;
    do
      ...
    end --pop
end --STACK
```

A copying routine may require that the object to be copied is not Void:

```
class ANY
...
feature
  copy( other : like Current ) is
    require
      non_void_other : not other = Void;
    do
      ...
    end -- copy
end -- ANY
```

The syntax for a precondition might seem slightly confusing at first. The basic syntax is the keyword **require** followed by one or more assertions. An assertion is a possible identifier followed by a colon ":" and then a Boolean expression. If more assertions are needed, a semicolon ";" is required:

```
...
feature
  remove_leaf is
    require
      non_empty_tree : not is_empty;
      non_void_tree : not tree = Void;
    do
      ...
    end --remove_leaf
```

Semantically, these two preconditions must both be met in order for the assertion to be met. Since they both must be met, it is equivalent to ANDing these two expressions in the body of our routine remove_leaf:

```
if (not is_empty) AND (not tree = Void) then
  precondition_met
  ... --rest of routine
else
  precondition_not_met
end;
```

The difference, though is that if we were to code the precondition in the body of *remove_leaf* as above, we would also be responsible for handling *precondition_not_met*. That is, we would have to devise a mechanism for alerting the caller of *remove_leaf* that our precondition was not met, and an appropriate action must be taken. Fortunately, Eiffel provides this capability for us.

When a precondition, or any assertion for that matter, is violated, an exception is generated in the class where the assertion is declared. We will discuss how to handle exceptions in a later section.

7.2.2 Post conditions

Assume that the caller of *remove_leaf* has met the preconditions. It is now safe to assume that the caller will expect nothing less than the removal of a leaf as the routine implies. A post condition guarantees this. More generally, it guarantees that once a precondition (if any) is met, then the routine will promise to fulfill a requirement which its client expects. We may now write *remove_leaf* as:

```
feature
  remove_leaf is
    local
      original_size : INTEGER;
    require
      non_empty_tree : not is_empty;
      non_void_tree : not tree = Void;
    do
      original_size := size;
      ...
    ensure
      one_less_element : size = original_size - 1;
  end --remove_leaf
```

Assuming that *original_size* stores the size of the tree at routine entry, and size is the number of elements at the time of the end of the routine, the post condition states that exactly one element was removed from the tree. The new *size* of the tree is exactly one less than it was before entering the routine, implying that an element was removed, as *remove_leaf* states.

7.2.2.1 Old Expressions

This doesn't mean outdated expressions! An old expression is denoted with the keyword **old** followed by an identifier or expression as in:

```
old size;
old arg1 + size;
old identification;
```

An old expression may only appear in the post condition of a routine. It denotes the value of its identifier upon routine entry. Above, **old** *size* refers to the value of *size*, when we entered the routine whose post condition this expression appears in. The usefulness of this may be shown by slightly modifying the remove_leaf routine:

```
class SOME_CLASS
feature
  Remove_leaf is
    require
      non_empty_tree : not is_empty;
      non_void_tree : not tree = Void;
    do
      ...
    ensure
      one_less_element : size = old size - 1;
    end --remove_leaf
end -- SOME_CLASS
```

This eliminates the need to declare an extra local variable, *original_size* as we had it, for the explicit purpose of temporarily storing the value of *size* at routine entry. An additional benefit of this keyword is the fact that it promotes readability. It is easier to read **old** *size* than go through the motions of determining exactly what *original_size* means.

7.2.3 Invariants

As computer scientists, we learn early how to formally specify the behavior of an algorithm. One of these mechanisms is the invariant, or a condition that must always hold true no matter what (actually, the invariant is allowed to be violated at certain critical times, but it must be restored).

Eiffel provides the ability to specify a series of invariants. This is done through the use of the **invariant** keyword. Syntactically, it must appear as the last construct in a class declaration:

```
class FIXED_DATA_STRUCTURE [T]
feature
...
invariant
  size >= 0;
  size <= 99;
end
```

This allows us to formally specify that at any given moment during the system's execution, the *size* attribute of an object declared as FIXED_DATA_STRUCTURE must be within 0 and 99 inclusively.

Another syntactical note is that the invariant clause may supply a descriptive identifier before its actual BOOLEAN expressions to further readability:

```
class FIXED_DATA_STRUCTURE [T]
feature
...
invariant
  stack_not_empty: size >= 0;
  no_overflow: size <= 99;
end
```

7.2.3.1 Looping Revisited - Variants and Invariants

There are two optional clauses of the **loop** construct, which was introduced in section 4.1, the **invariant** and the **variant**. These clauses provide support for Eiffel's contract programming ties.

The i**nvariant** clause states that upon each iteration of the loop body, the BOOLEAN expressions in this clause must hold true. This is useful from both a design as well as a implementation perspective. From a design point of view, **invariants** help specify loop correctness- conditions that must hold true at all times. From an implementation point of view these expressions, if they happen to evaluate to *False* will enable a

programmer to easily handle this breach with Eiffel's exception handling mechanisms (discussed in section 7.4.1).

To illustrate the use of invariants and variants, and how they work, we present this simple **loop** which outputs written pages to a printer:

```
from
  num_pages := pages_to_be_printed.
until
  num_pages = 0;
invariant
  printer.on_line;
variant
  num_pages;
loop
  ...
  printer.output_next;
  ...
end
```

The **loop** begins with initialization, denoted by the **from** keyword. Here *num_pages* is set to the number of pages to be printed, *num_pages_to_be_printed*. Next, the stopping condition, denoted with the **until** keyword, is tested. If it evaluates to true (the number of pages that are to be printed is zero), then the loop body is not executed. The stopping condition will be evaluated prior to each iteration of the loop body.

The new clauses introduced here are the **invariant** and **variant**. The **invariant** clause states that prior to each iteration of the loop, the BOOLEAN expression(s) here must evaluate to *True*. If during any test of the **invariant**, the expression evaluates to *False*, then an exception will be raised, and it may be handled as described later. Prior to the first iteration of the **loop** body, the **invariant** is tested for the above conditions.

The **variant** on the other hand is a little different. The type of its enclosing expression(s) is INTEGER. Its expressions are evaluated once, prior to the first execution of the **loop** body, and tested prior to each iteration (including the first iteration). In the above text, the **variant** is *num_pages* which is the number of pages to be printed. If for instance it is initially evaluated to be 3, then prior to the next iterations, it is automatically decremented to 2 and then to 1 and so on.. This guarantees that the **loop** will terminate at some finite point in time due to the following rule: if the variant ever reaches a negative value (by its automatic decrementing) then an exception is raised, and the loop will terminate.

The **invariant** and **variant** are language mechanisms that allow easy specification (in source code!) of loop correctness. The benefits of this can be found in just about every situation where a looping construct is called for.

7.2.4 A Note on Efficiency

At this point one begins to realize that Eiffel's assertions may cause more harm than good in certain circumstances. Notably system performance, which is often a major factor in graphic-intensive or real-time applications, may be muffled due to the overhead of assertions. In *Eiffel: The Language* which is the formal reference for the Eiffel language, it describes several user-selectable possibilities for run-time assertion monitoring. One of them being no assertion checking of any kind. This means that no classes will be monitored for assertions. The assertions include preconditions, post conditions, and invariants.

It is also possible to define for a class C that a certain level of assertion monitoring be done, and for another class D, a different or no monitoring be done. This empowers the developer to develop a class with assertion monitoring turned on, and once it has been fully tested and can be guaranteed within a reasonable degree, to work, assertion monitoring may be shut off.

7.3 The New VEHICLE class

Now that we have an understanding of the role of assertions, it is now time to modify our VEHICLE class to support assertions. This will tighten the security around objects of type VEHICLE by not allowing dangerous arguments to be passed into routines.

```
class VEHICLE

creation start;

feature {NONE}
  color, num_passengers : INTEGER
    -- "private" data exported to NONE

feature
  Red, Blue, Green : INTEGER is unique;
        -- constants available to all clients

  start (new_num_passengers : INTEGER) is
  require
    new_num_passengers >= 1;
  do
    color := Red; --default
```

```
    num_passengers := new_num_passengers; --default;
    io.put_string ( "Hi!. I'm a new VEHICLE" );
    io.new_line;
  ensure
    color = Red;
    num_passengers = new_num_passengers;
  end; --start

  set_color( new_color : INTEGER ) is
  require
    new_color >= Red;
    new_color <= Green;
  do
   color := new_color;
  end --set_color

  set_num_passengers( new_num_passengers : INTEGER ) is
  require
    new_num_passengers >= 0;
  do
    num_passengers := new_num_passengers;
  end --set_num_passengers

  get_color is
  do
    Result := color;
  end --get_color

  get_num_passengers is
  do
    Result := num_passengers;
  end --get_num_passengers

  get_id is
  do
    Result := object_id;
  end --get_id

  invariant
    last_attrib >= 0;

end --VEHICLE
```

The new VEHICLE class is now much safer than its original version. This is because each routine has been fitted with a precondition that mandates that all arguments

be within reasonable bounds. For instance, any client who wishes to change the *color* of a VEHICLE object must pass as argument a value that is within the *Red* to *Green* range. Also, a change in the number of passengers via the *set_num_passengers* routine can not specify a negative number of passengers. We could also have required that one not attempt to set a ridiculously high number of passengers by specifying a limit to the *new_num_passengers* argument. This additional constraint may be added if the reader wishes to do so. We have chosen not to do so.

Now that we have the appropriate constraints in place in our routines and our class, we must now deal with the situation where one of these conditions is violated. Consider:

```
class DRIVER
feature
  car : VEHICLE;
  some_routine( some_arg : SOME_TYPE ) is
    do
      ...
      car.set_color( -2 );  --this is invalid!!
      ...
    end --some_routine
end --DRIVER
```

The call to *set_color* with the -2 argument is clearly a violation of *set_color*'s precondition. There is no dispute about this. The question is then, what happens now?

7.4 Exceptions

"If a routine executes a component and that component fails, this will prevent the routine's execution from proceeding as planned; such an event is called an **exception**" [Meyer, 1992]. Exceptions in Eiffel are essentially messages that tell a routine that a failure to meet a requirement has occurred. The exception message itself is generated in the routine where the violation occurred. Set_color's precondition was violated when some_routine attempted to pass a negative argument. Since it was set_color's precondition that was violated, set_color will receive the exception message initially. It is up to the implementor of set_color to decide how to handle the exception.

7.4.1 Handling Exceptions - Rescue clauses

It is possible to specify in Eiffel, how a routine is to handle an exception, should it ever be passed an exception message. This is done through the rescue clause which begins with the keyword **rescue**. A routine which has a rescue clause is allowed to specify

appropriate actions that should be taken to handle the exception. A sample of the rescue clause in action is now given:

```
feature some_feature( some_arg : SOME_TYPE ) is
require
  some_arg >= 0;
do
  some_attribute := some_arg;
ensure
  some_attribute = some_arg;
rescue
  error_code_attribute := 1;
    --raise internal error flag for debugging
end;
```

Notice the rescue clause at the bottom of the routine. This is the only place a rescue clause may appear- as the last clause of a routine. The clause here states that if the routine body

```
do
  some_attribute := some_arg;
```

fails to execute properly, or if the post-condition

```
ensure
  some_attribute = some_arg;
```

is not met, appropriate action should be taken. That action being the internal flag *error_code_attribute* be set to *some_arg*. The author could also have chosen to handle an above exception differently as seen fit.

There is generally no right or wrong way to handle exceptions. The actions taken in a rescue clause should be based upon how severe the exception is, and how the programmer wishes to react to an exception.

7.4.2 Retry Commands

Another possible venue concerning handling exceptions is to attempt to return the object to a stable state, and possibly re-execute the routine body. We will demonstrate this with an analogy- making a telephone call. If after dialing a telephone, we get a busy signal, it is not necessary at all to give up attempting to complete this task, swallow our losses, and continue with life. In fact it is probably easier just to wait a few minutes and try again later.

The retry command helps support Eiffel's close ties with contract programming. If a routine is unable to meet its post conditions, it may not be enough to accept losses and notify the callers of that routine. It could be plausible to execute the routine once again in an attempt to meet the post-condition.

To see how a retry command is actually used, we present this example:

```
class PHONE_LINE
feature
  dial( number : STRING ) is
  do
    modem.call( number )
  ensure
    modem.is_connected;
  rescue
    time_out; -- wait a random period of time
    retry;
  end;
end;
```

It should be noted that the only place a retry command can be found is in a rescue clause.

The presence of a rescue clause with or without a retry statement does not imply that a post-condition is required. It is not necessary to have a post-condition in a routine, even if a rescue clause may be needed:

```
feature
  SOME_FEATURE is
  do
    ...
  rescue
    -- some action
  end -- SOME_FEATURE
...
```

7.5 LAND_VEHICLE and WATER_VEHICLE

This section will introduce two new classes LAND_VEHICLE and WATER_VEHICLE both descendants of class VEHICLE as presented above. We will give the text for LAND_VEHICLE in a moment; WATER_VEHICLE will soon follow.

The basis of this section is to allow the reader to witness and participate in the inheritance mechanism used in a tangible example. At this point only single inheritance will be employed; multiple will be used in a later section when we develop a HYDRO_LAND_VEHICLE based upon WATER_VEHICLE and LAND_VEHICLE.

The specifications for our LAND_VEHICLE is as follows:

- The class must provide a facility that enables each instance to be uniquely identifiable.
- Attributes:
  Color
  Number of passengers
  Number of Wheels
- Routines:
  - Routines to set and retrieve the above attributes
  - A constructor routine is also required. It must accept a new Engine Type attribute as argument.

The first thing that the reader will notice is the similarity between this specification and the specification we already have for the VEHICLE class. In fact a good portion of software utilities, data structures and algorithms are patterned after each other (not necessarily to the degree that LAND_VEHICLE and VEHICLE are similar).

This similarity puts us in a perfect position to use inheritance. Since most of the specification of LAND_VEHICLE has already been implemented in VEHICLE, we can inherit from VEHICLE and use the features previously described to adapt a new LAND_VEHICLE class. We see this class being implemented as such (again, individual interpretation is left up to the reader).

```
class LAND_VEHICLE

creation start;

inherit
  VEHICLE
    redefine start;
  end;

feature {NONE}
  num_wheels : INTEGER;

feature
  start (new_num_passengers : INTEGER) is
  require
```

```
    new_num_passengers >= 1;
  do
    num_wheels := 4;
    color := Red;
    num_passengers := new_num_passengers;
    io.put_string ( "Hi! I'm a new LAND_VEHICLE" );
    io.new_line;
  ensure
    num_wheels = 4;
    color = Red;
    num_passengers = new_num_passengers;
  end; --start

  set_num_wheels (new_num_wheels : INTEGER) is
  require
    new_num_wheels % 2 = 0; --even number of wheels
    new_num_wheels >= 2; --at least two wheels
  do
    num_wheels := new_num_wheels;
  ensure
    num_wheels = new_num_wheels;
  end; --set_num_wheels

  get_num_wheels : INTEGER is
  do
    Result := num_wheels;
  ensure
    Result = num_wheels;
  end; --get_num_wheels;
end --LAND_VEHICLE;
```

The new features introduced here are *num_wheels* which is visible to no clients, *get_num_wheels* and *set_num_wheels*, both visible to all clients. This implementation fulfills our requirement for the LAND_VEHICLE specification. The features not directly implemented in this class, such as *color* have been inherited from VEHICLE. Notice the smaller size of the LAND_VEHICLE class as compared to VEHICLE. Yet, in terms of functionality, LAND_VEHICLE is more advanced and useful. This is the major benefit of object-oriented programming- reuse. The ability to use existing code reduces largely the cost of future projects implemented. In order to take full advantage of this capability, it is necessary that a language provide mechanisms that capture these abilities and provide a safe and easily readable syntax for the reader (programmer) to use. Eiffel does just that.

We now continue our venture in developing a small hierarchy of classes by giving the (informal) specification of WATER_VEHICLE:

- The class must provide a facility that enables each instance to be uniquely identifiable.
- Attributes:
  - Color
  - Number of passengers
  - Weight capacity
- Routines:
  - Routines to set and retrieve the above attributes
  - A constructor routine is also required. It must accept a new and valid number of passengers as an argument

The text for this new class is given below:

```
class WATER_VEHICLE

creation start;

inherit
  VEHICLE
    redefine start;
  end;

feature {NONE}
  capacity : INTEGER;

feature
  start (new_num_passengers) is
  require
    new_num_passengers >= 1;
  do
    num_passengers := new_num_passengers;
    color := Red;
    capacity := 500;
    io.putstring ( "Hi! I'm a new WATER_VEHICLE" );
  ensure
    num_passengers = new_num_passengers;
    color = Red;
    capacity = 500;
  end; --start

  set_capacity (new_capacity : INTEGER) is
  require
    new_capacity > 0;
  do
    capacity := new_capacity;
  ensure
```

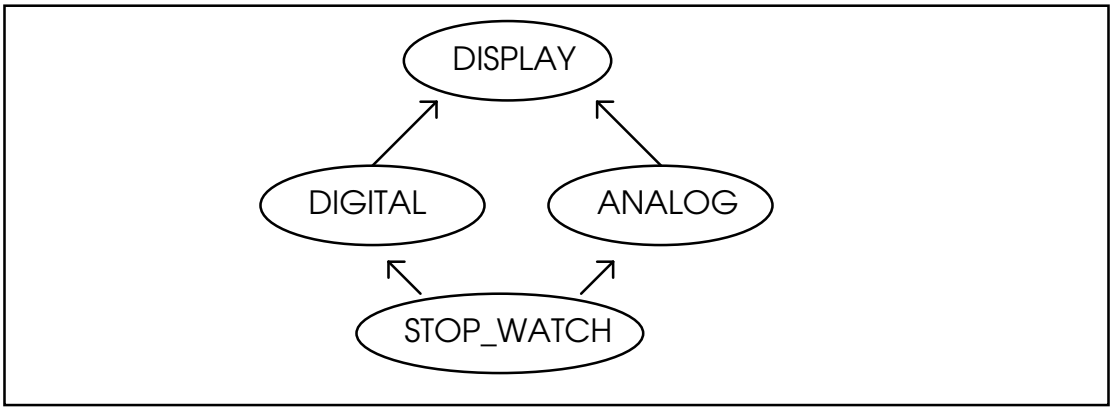
```
      capacity = new_capacity;
    end; --set_capacity

    get_capacity : INTEGER is
    do
      Result := capacity;
    ensure
      Result = capacity;
    end; --get_capacity
  end --LAND_VEHICLE
```

We have thus far constructed three of the four classes we wish to describe. It is now left to describe and implement the fourth class, HYDRO_LAND_VEHICLE. As the reader has probably already figured out, this class will inherit from WATER_VEHICLE and LAND_VEHICLE. This may give rise to some questions concerning the features from VEHICLE that are implicitly inherited twice (repeated inheritance) through WATER_VEHICLE and LAND_VEHICLE. We will now discuss this situation.

7.6 Repeated Inheritance

A class which utilized multiple inheritance may inherit, implicitly, from a particular parent more than once (through different routes). Consider:



Here, STOP_WATCH inherits from DIGITAL and ANALOG, each of which inherit from DISPLAY. What happens to the features of DISPLAY that exist both in DIGITAL and ANALOG? Are two copies inherited in STOP_WATCH; or is just one copy inherited?

The answer is that Eiffel will resolve any repeated inheritance ambiguities by only allowing one copy of all features to end up in a descendant class. In the above case, all of the features of DISPLAY that are inherited through DIGITAL and ANALOG are only inherited once in STOP_WATCH. For instance, if DISPLAY provided an *am_pm* feature,

only one such feature would wind up in STOP_WATCH even though it could potentially be duplicated via DIGITAL and ANALOG.

7.7 HYDRO_LAND_VEHICLE

Referring to our VEHICLE classes, a similar situation has arisen since it is now time to write our HYDRO_LAND_VEHICLE. Since HYDRO_LAND_VEHICLE will inherit through LAND_VEHICLE and WATER_VEHICLE, what happens to the features of VEHICLE that could possibly be duplicated via LAND_VEHICLE and WATER_VEHICLE? The repeated inheritance discussion above should give the reader an answer.

One question that might enter one's mind is what happens to the *start* feature. That is, it was redefined in LAND_VEHICLE and WATER_VEHICLE, does this mean that it will be inherited twice since it was redefined? The answer is: we have encountered a name clash.

Since *start* was redefined in LAND_VEHICLE and WATER_VEHICLE, it no longer comes from VEHICLE, it now comes from both of its descendants, in two versions. Recall from chapter 5 that when two features with the same name, here *start*, is inherited from two classes, a name collision, or clash occurs. Eiffel provides several options for this situation. We have chosen the following:

```
class HYDRO_LAND_VEHICLE
creation start;
inherit
  LAND_VEHICLE
    undefine start;
  end;
  WATER_VEHICLE
    undefine start;
  end;

feature
  start is
  do
    io.putstring( "Hi! I'm a new HYDRO_LAND_VEHICLE!" );
    io.newline;
  end; --start
end --HYDRO_LAND_VEHICLE
```

Our choice was to undefine **both** of the *start* features inherited from our ancestors. We then were able to totally free up the name *start* in our new class, and use it as we choose. We have decided to let it keep its reputation as a creation routine.

We now have a fully functioning classification for vehicles that can travel both on land and water. Notice that it took virtually no extra implementation on our part because the capabilities were already created in our ancestor classes. We just built on what they have provided and created an entirely new classification. This is one of the main foci of the object-oriented paradigm- reusability.

With these last words, we come to an end of this section. In the next chapter, we will discuss Eiffel's advanced mechanisms that make the language suitable for analysis and design. With this, Eiffel can provide a smooth and seamless transition all the way through implementation.

Chapter 8:

Eiffel's Role in Analysis and Design
Final Words Concerning Eiffel's Future

8.0 Introduction
8.1 Deferring Classes and Features
8.2 Explicit Creation
8.3 What Do I Tell My Clients?
8.4 The Future of Eiffel

8.0 Introduction

Writing software is becoming more and more an action of sitting down and thinking out a problem through and through with less time being spent on actually *writing* the software. This emphasis on analysis and design is being promoted by the object-oriented paradigm as reflected by its radically different metric estimates. In a book entitled "Object-oriented Software Construction" by Bertrand Meyer, it is estimated that up to 70% of software costs can come from maintenance. That is, once a product is actually "complete", it must be cleaned up and continuously updated, modified, etc. If the underlying system has not been properly and thoroughly documented, analyzed, and designed extensively, the post-release period can become critically costly.

Emphasis is now being emphatically placed more and more on initial efforts. This new attitude has dual benefit: (1) objects created for project *i* can be reused more readily in projects *i+1*... thus reducing the costs of future projects, and (2) the potentially 70% cost of maintenance is dramatically reduced and placed in earlier stages of the software cycle where less risk and more flexibility concerning a design is enabled.

Eiffel is suitable as a PDL (Program Design Language). PDL's are languages that are used to describe software without being concerned with a particular implementation. It is almost a blessing and a curse that Eiffel is this suitable. It is a blessing because when software that needs to be implemented, is modeled or described using Eiffel, the implementation is not even needed. It has already been done in the modeling! On the other hand, using Eiffel as a PDL might be too tempting to some designers in that the mechanisms provided by Eiffel, especially concerning its extensive feature adaptation mechanisms, do not appear in most other languages. How do we implement a model when languages do not support our notations?

8.1 Deferring Classes and Features

The first step Eiffel takes towards being a suitable PDL is its ability to capture the existing knowledge of an object or group of object without actually having to fully implement it. We can create a partially completed class that reflects our existing knowledge of the system and be able to inherit more and more specialized classes that better reflect our information. This may be done as follows:

```
deferred class FILE_PROCESSOR
feature {NONE}
  file_to_be_processed : FILE;
feature {ANY}
  process is
    deferred
  end; --process
end -- FILE_PROCESSOR
```

This class might reflect the fact that we **know** that we are going to process files, but we are not quite sure how this will be done and on exactly what kinds of files. Let's say that later on during analysis, we determine that we will be processing vector graphics files to determine how many lines appear in a file. The existence of this new knowledge should not be left in the minds of a select few analysis personnel. It should be documented.

```
class VECTOR_PROCESSOR
inherit
  FILE_PROCESSOR
    rename file_to_be_processed as vector_file;
  end;
feature
  process is
  do
    ...
    -- class text left out
  end; -- process
end --VECTOR_PROCESSOR
```

This new class allows us to say that new information has been gathered concerning the FILE_PROCESSOR classification. In essence we are giving form as well as function to our predecessor which manifests in VECTOR_PROCESSOR.

One might question why we did not have to **redefine** our *process* routine. It existed before in FILE_PROCESSOR, but we also declared it in VECTOR_PROCESSOR. Is this legal? Since it was indeed declared in FILE_PROCESSOR, but not defined, that is implemented, we could not redefine something which has no definition. When we inherit a feature which itself is deferred from a deferred class, we are **effecting** it, not redefining it. Giving meaning to routines that can only be partially defined in previous classes is an attractive capability that Eiffel provides.

In order to qualify as a deferred class, at least one feature of that class must be deferred. In which case, the class header must start with the **deferred** keyword. Deferred classes can not be instantiated. That is, no objects can be created of that type. They can be declared of a deferred class, but not created.

8.2 Explicit Creation

What happens though, when we know that a class is deferred, but we need to declare an entity of that type somewhere else. For instance, if we are developing a file management system, one of the classes might contain special services that process various files- graphics, texts, comma-delimited, etc. Consider:

```
class FILE_UTILITIES
feature
  process : FILE_PROCESSOR:
  ...
end --FILE_UTILITIES
```

Since we know that we need a file processing class in our utilities, we know that a declaration of this type (or similar) will appear at some stage of the development process. Can we do this? Yes. Eiffel permits entities to be declared of a deferred class, but not instantiated. How, then can we instantiate a class of a type other than that of which it is declared? The answer is explicit creation. Observe: if somewhere in our FILE_UTILITIES we find out that we will be needing services from the VECTOR_PROCESSOR class, we may instantiate the *process* entity of the VECTOR_PROCESSING class as follows:

```
! VECTOR_PROCESSOR ! process_file
```

This means that even though *process* was declared to be a FILE_PROCESSOR, it may be created (instantiated) as a VECTOR_PROCESSOR. This mechanism provides tremendous flexibility with respect to supporting an initial analysis and being able to actually use a class, even though its description might be **deferred**. Explicit creations may only instantiate an entity as a type which is a descendant of its actual declared type. Above, since VECTOR_PROCESSOR is a descendant of FILE_PROCESSOR, no error will be generated by a compiler.

8.3 What Do I Tell My Clients?

During the course of reading this, one might observe a property of Eiffel that might seem disturbing at first. In most traditional languages, it is customary to write two distinct files when implementing a class. A definition (header), and an implementation module. The definition module is what is given to a client who wishes to use the software one might have implemented in the implementation module. How does Eiffel provide a programmer with the capability to describe for a client what features are available? It doesn't!

The language itself provides no mechanism for accomplishing this. And rightfully so, as we will demonstrate. In the official description of Eiffel, "Eiffel: The Language" (ETL for short) by Bertrand Meyer, it is suggested that a language processing tool be provided to accomplish this. This tool should be given a full class text; its job is to output a **flat** form of that class. In this **flat** form, only the essentials of a class are provided. These include the class name, whether it is deferred or expanded, all features and to whom they are available, preconditions, post conditions of all routines, and any class invariants that

may exist. This flat form includes all of the inherited features of the class plus the features introduced by that class.

A second form called the **flat-short** form will output similar results as the flat form, but it will only do this for features that the class itself introduces, excluding inherited features.

These forms, whose formats are fully documented in ETL, make it easy for a programmer to concern him or herself with the implementation details of the system to be constructed while eliminating the hassle of writing essentially redundant information about a class.

8.4 The Future of Eiffel

In this chapter we have shown how Eiffel may be used during analysis and design, and how it provides mechanisms for smoothly transitioning to implementation. This last chapter has now come to an end as has this paper. We hope that we have successfully shown Eiffel's importance and its advancements concerning software engineering. The efforts put forth by those who are supporting Eiffel currently should not go unmentioned. We have reserved a place for this group here.

The group of people who are responsible for the future of Eiffel are collectively known as NICE (I am sure they really are nice, too!). NICE is the Non-profit International Consortium for Eiffel. It is a group solely devoted to the technical aspects of Eiffel as opposed to its commercial aspects. Right now, faithful followers of Eiffel (the authors included) hope that this consortium will do everything in its power to keep Eiffel as concise, simple and powerful as it is now, so that it will not fall prey to the inundation of redundant additions by organizations and the like that have corroded more recent languages.

The future of Eiffel is open and bright. If the right people are put in positions where important decisions concerning the language can be made intelligently and wisely, then Eiffel should (we hope) solve the problems of the software industry with one swift swoop.

Its infinite possibilities will nonetheless be the attention of the authors for the near and far future. We hope the reader will participate with us, and its many followers, in this exciting endeavor...