
Into Eiffel

by Ian Joyner

Harmonic Systems and Object Tools *i.joyner@acm.org*

This is a short introduction to Eiffel. The object is to draw your attention to the salient features of Eiffel; from this framework you can build up a good knowledge of Eiffel from many other excellent sources. You can also find introduction papers that give more detail at <http://www.eiffel.com/> and <http://www.elj.com/>. The many books on Eiffel give even more detail, up to *Eiffel: The Language*, which is the complete language definition. *Object-oriented Software Construction* is the classic text on object-oriented programming; it explains the philosophy of OO and introduces Eiffel as its notation. Details about these books and others can also be found at <http://www.eiffel.com/>.

Classes

The basic construct in Eiffel is the **class**. In fact there is no other construct of note. So immediately we have a language which is fundamentally simpler than C/C++ and Object Pascal.

A **class** represents entities, the attributes of those entities and the operations that those entities can perform. This gives you a fundamental mechanism for project organisation because any application is organized into a set of interacting classes.

Classes represent real world entities in a model, but can represent more artificial artifacts that occur only in computer programs. An example of an Eiffel class is:

```
class
  CAR
end
```

A class represents all objects of that type. For instance, in the real world we have one concept of *CAR*, but there are many instances of *CARs*. We should be careful to distinguish between a class as a conceptual design pattern of entities and objects that represent the entities themselves.

Libraries

One of the greatest promises of OO is reuse; that is, you should not have to rewrite many basic concepts that frequently appear. Classes that frequently appear are organized into libraries, such as EiffelBase for many basic types such as *INTEGER*, *REAL*, *STRING*, etc. Many collections are also in the base library such as *LISTs* and *ARRAYs*, with many others.

Not only do you have the EiffelBase (sometimes called Kernel and ELKS) library, but many other libraries to interface to databases, CORBA, etc. Also on the Macintosh with EiffelS for CodeWarrior, you get the MacOS Toolbox Eiffel Library, Eiffel Carbon, which is a full OO interface to the MacOS. Carbon can simply be used as an Eiffel API to MacOS, but with a single call, it becomes a full event-driven application framework like MacApp and PowerPlant, making application development even simpler. You will find Carbon cleaner and easier to understand and find what you want than any framework written in C++, and it is the consistency of the Eiffel language that enables this. Carbon also follows uniform naming patterns that many other Eiffel libraries vendors have adopted. This means that the programmer has less to remember and has to consult API references less frequently.

Libraries are not defined as part of the Eiffel language—they are simply collections of related classes.

Features

Each class has a set of features that represent the attributes and operations of a class. Operations on an object typically alter the state of the object, that is, will change the values of one or more of its attributes. In Eiffel, such operations are known as **procedures**. Fields in an object can be changed only by routines in the object.

Functions are computations that return an answer to a query about the state of an object, but do not change the state of an object (although this is by convention, rather than being enforced in Eiffel). Computational functions and procedures are together known as **routines**. Functions, however, have more in common with reading an attribute field or even constant than with procedures. When performing a query, it does not matter to the querier whether the answer is retrieved from a pre-computed field of the object or by a computation that is done at that moment. Eiffel needs no low-level call operator such as ‘()’ to invoke functions. This makes Eiffel programs very flexible because functions with no arguments can be redefined as fields or constants.

For example, the attribute speed of a *CAR* object could either be stored in a field or computed from other inputs or fields stored in the object. (Indeed reading a field involves a computation of retrieving the contents of that item; this computation is in fact hidden to the programmer, which leads many to think that field access and functions are different, but really field or constant access is functional access. This is called the *Principle of Uniform Access* and makes software very flexible.)

Attributes themselves can be *fields* or *constants*. A field sets aside space within each created object where the value is stored. A constant is compiled into the code, and does not use up space in an object.

To summarise: routines are either procedures or functions; attributes can be fields, constants or functions. These four entities, procedures, functions, fields and constants are collectively known as features.

An example of a class with features is:

```
class
  CAR
feature
  colour: COLOUR -- a field
```

```
velocity: INTEGER is -- a function
  do
    Result := speed
  end

wheels: INTEGER is 4 -- a constant

speed: INTEGER

stop is -- a procedure
  do
    speed := 0
  end
end -- CAR
```

Note that the features of a class are introduced by the keyword **feature**. A number of other points can be seen about Eiffel style and syntax. Comments are introduced by -- (like an em-dash, —). This indicates a comment to the end of line. Grouped entities are terminated by the keyword **end**. Eiffel has no **begin** keyword since it is superfluous. (Small syntactic problems like not putting a semicolon before an **else** as in Pascal are avoided in Eiffel. Remember that semicolons are merely typographical for human readability in Eiffel and therefore optional. Code in in fact more readable without semicolons.) Also superfluous are semicolons, but these may optionally be placed between constructs. Eiffel has a clean and modern syntax making programs much easier to read.

The features *speed* and *velocity* can be interchanged as queries. In fact, this is an artificial example to show only the differences and similarities between functions and fields.

In Eiffel style, keywords are shown in **bold** and user named entities in *italics*. Class names are given in uppercase, which follows the mathematical typographical convention for types. Eiffel is not case sensitive, so it is up to the programmer to follow style conventions. Entity names made up of more than one word separate the constituent words with underscore ‘_’, for example *SPEEDY_CAR*. (As you can see, the underscore style is more consistent than the capitalization style, SpeedyCar, which does not work for upper case type names, or constant names as in C style. The underscore style is also more readable, giving a better approximation of white space in between words.)

Note also in your editor, you will not have to enter words in bold and italics—these are done by formatting software.

C and C++ programmers might be wondering how to make features public, protected and private. With Eiffel you have far more control; any set of features introduced by the **feature** keyword can be exported to other specific classes. Thus you have the possibility of many shades of grey between public and private. You might want a feature to be public to some specific classes, but private to others. This also covers the friend mechanism of C++ since a special implementation relationship can be specified between certain classes.

In Eiffel there are two specific classes *ANY* and *NONE*. *ANY* is at the top of the inheritance hierarchy and exporting to *ANY* is equivalent to public. *ANY* is automatically inherited by all classes. It is like Object in Java, and TObject in MacApp. *NONE* is at the bottom of the inheritance hierarchy and exporting to *NONE* is the equivalent of protected in C++. There is no strict equivalent of private, as Eiffel believes it is not sensible to restrict visibility in subclasses. We will not look at the specifics of export since this is covered in longer tutorials and this is meant to be a short tutorial.

Inheritance

Perhaps the distinguishing feature of object-oriented languages—after classes—to many people is inheritance. Simply put, inheritance is the ability to create a new class from an existing class. It only makes sense to do this when the existing class provides features wanted in the new class.

Where features are not exactly what is required in the new class, they can be redefined. Redefinition can take two forms—either redefinition of an operation or function, or redefinition of type. Redefinition of operation is only meaningful where the feature is a routine. Redefinition of type can be applied to fields, functions, and arguments

In order to build new classes out of existing classes and to reuse features already defined in those classes, you use inheritance. In Eiffel, you use the inheritance clause as follows:

```
deferred class  
VEHICLE
```

```
feature
  velocity: INTEGER is -- a function
    do
      Result := speed
    end

  wheels: INTEGER is
    deferred
  end

  speed: INTEGER

  stop is -- a procedure
    deferred
  end
end -- VEHICLE
```

```
class
  CAR
inherit
  VEHICLE
feature
  colour: COLOUR -- a field

  wheels: INTEGER is 4 -- a constant

  speed: INTEGER

  stop is -- a procedure
    do
      speed := 0
    end
end -- CAR
```

This is a very simple example of inheritance—it only shows single inheritance. Eiffel has multiple inheritance. As those who have used a language with multiple inheritance know, if two features with the same name are inherited from two different classes, a clash occurs. Eiffel solves this by having a **rename** clause that allows you to rename one or both of the features to remove the clash. This is different to

the scope resolution operator `::` of C++, where disambiguation must be done on every reference to the clashing inherited members.

The example also does not show how to redefine a feature. If you wish to redefine a feature, you must put a **redefine** clause in your inheritance clause. The example also shows deferred features. You do not have to put a redefine clause in order to give these a definition in a subclass. Defining a deferred feature is called *effecting* that feature. Our example shows two deferred features, *stop* and *wheels*. Note that *stop* is effected as a routine, whereas *wheels* is effected as a constant.

Apart from **rename** and **redefine** clauses in the inheritance clause, you can change the export status of inherited features with the **export** clause (you should not use the export clause to restrict access that was previously granted in a parent—that would be mean, but it has some negative theoretical type considerations). Two other clauses that give complete control over inheritance are the **undefine** and **select** clauses. Thus, when inheriting any class, you can control the inheritance with the five subclasses: **rename**, **export**, **undefine**, **redefine** and **select**.

Many object-oriented languages do not provide redefinition of types and thus are no variant, but this means that many things that can be easily achieved by simple type redefinition are not possible and must be done by providing extra code and classes. In languages that have redefinition of types, the redefinition can be covariant or contravariant. Covariant means the new type must be a subtype of the original type; contravariant means the new type must be a supertype of the original.

Covariant redefinition of fields and functions provides no problems, but covariant redefinition of arguments does create a problem that illegal types can be passed as arguments. However, this is the most practical way to redefine types. Consider that the type of a field can be changed in a subclass:

```
a_field: A_TYPE  
  
set_a_field (to_field: A_TYPE) is  
  do  
    a_field := to_field  
  end
```

in a subclass these could be redefined as follows:

```
a_field: B_TYPE
```

```
set_a_field (to_field: B_TYPE) is  
  do  
    a_field := to_field  
  end
```

note that the type of the `to_field` argument has followed the type of the type of the field. This sets the practical precedent for covariants of arguments. Now Eiffel does even better than this—in the second class, only the type of the argument has changed but the code is exactly the same and needlessly repeated. To deal with this very common situation, Eiffel introduces anchored types:

```
a_field: A_TYPE  
  
set_a_field (to_field: like a_field) is  
  do  
    a_field := to_field  
  end
```

and in the subclass all you need to do is redefine the field's type:

```
a_field: B_TYPE
```

and no redefinition of the set routine is needed. Using anchored types, most uses of covariant arguments can be avoided. Another point in Eiffel is that accessor or `get_` routines need not be provided since fields can be accessed functionally by the principle of uniform access. C programmers argue that this breaks the principle of data hiding—in C accessing fields directly exposes the structure of the class, but in Eiffel it does not, the principle of data hiding being subsumed by the cleaner principle of uniform access.

The other important facility that Eiffel provides is multiple inheritance. Most people understand the use of single inheritance where a subclass gains characteristics from a single parent class. In some cases this can be generalised to multiple inheritance where a subclass can gain characteristics from multiple classes.

Some people see problems with multiple inheritance since classes designed in a single inheritance environment tend to become large and unwieldy, covering multi-

ple concepts. Multiple inheritance allows finer-grained design of classes with a single concept per class which can be combined to form new classes. Thus much more powerful object-oriented design can be accomplished with multiple inheritance.

A problem that multiple inheritance introduces is ambiguity. Suppose the same feature is inherited from two or more classes with slightly different implementations. Which is the correct one to invoke when a client calls it?

Eiffel resolves all such ambiguities in its inheritance clause, by renaming and selecting inherited features. For example

```
class
  PLANE
inherit
  VEHICLE
  rename
    drive as taxi
  end
  FLYING_OBJECT
  select
    move
  end

feature
  ...
end -- PLANE
```

The inheritance clause in Eiffel can also be used to export existing features to more classes, undefine features, and redefine features.

Since ambiguity is completely resolved within the inheritance clause in Eiffel, this is a fundamentally different approach to C++, where the client programmer must be aware of the nature of multiple inheritance in a class being used and resolve the ambiguity in every access to that class. If the class changes then the potential to have to change all the client classes exists. Because of this messiness in C++, many people believe that multiple inheritance is per se messy, but this messiness does not exist in Eiffel—any headaches in resolving problems due to multiple inheritance must be handled by the creator of the class, not left to clients.

Another aspect of inheritance is that either interfaces or implementations, or both can be inherited. In languages such as Java, only interfaces can be multiply inherited—implementations can be inherited only from one class. Java does this in order to avoid the ambiguity problem that we have seen is elegantly solved in Eiffel.

The problem with Java's scheme of restricting inheritance of implementation is that interfaces must be implemented in the classes that inherit them. This gives the possibility that the same code must be implemented in many subclasses. Often just a default implementation is required between many classes, and this can be put in an appropriate ancestor class with implementation inheritance. Thus full multiple inheritance of implementation is required for practical programming—arguments against it are purely theoretical.

In Eiffel, there is no separate concept for interface as in Java, everything is included in the class concept. To overcome some of the shortcomings of not having true multiple inheritance in Java, inner classes can be used to provide default implementations, but this is just adding complexity in other areas. In Eiffel, if you are really convinced that single inheritance of implementation and only interfaces can be multiply inherited, you can program in this style—you won't want to follow that rule for long before it becomes apparent that Java's interface style of multiple inheritance is a burdensome restriction.

Genericity

Inheritance is one of the fundamental mechanisms for reuse—so is genericity. Genericity is also important in making programs type safe without resorting to type casts. Java does not have genericity, and many type casts are needed to make up for this deficiency—this is burdensome to the programmer. C++ has genericity in the form of template classes. If you have had problems understanding C++ templates—don't worry—Eiffel's generic syntax is much easier and more powerful because it also allows generic parameters to be constrained; this is known as *constrained genericity* (also known as *bounded-* and *F-bounded polymorphism*).

In order to use genericity you create a generic class with formal generic parameters. These are generic types where the type is left open to be instantiated by actual generic types. Generics are most useful in collection classes. For example, *LISTs* can store *INTEGERS*, *ANIMALS*, and other objects. Thus the *LIST* class is declared as:

```
class LIST [T]
...
end
```

The actual lists are instantiated as:

```
il: LIST [INTEGER] -- LIST of INTEGERS
animal_list: LIST [ANIMAL] -- LIST of ANIMALs
list_list: LIST [LIST [INTEGER]] -- LIST of LISTs of INTEGERS
```

A generic class can also restrict the kinds of actual parameters. For example:

```
class SHELF [ITEM -> SHELF_ITEM]
...
end
```

Here any actual generic type must be a *SHELF_ITEM* in order to instantiate a valid shelf.

The key point to remember about generics is that they allow you to write general algorithmic patterns that apply to a variety of types. The variety of types can be restricted with constrained genericity, where the genericity is known to work only on certain types. Where the generic types are not constrained, the algorithmic pattern is universally applicable.

Object Creation and Garbage Collection

Objects are created with the special **create** instruction. An example looks as follows:

```
c: C
create c.make
```

or

```
create {D} c.make
```

In the first example, an object of type *C* is created and attached to the reference *c*. (Remember Eiffel is case insensitive, but the name *c* here is used for a variable and *C* for a class type, but there is no name clash.) In the second example, an object of

type *D*, where *D* conforms to *C* (that is *D* is a subclass of *C*), is created and attached to *c*.

The other point to note is that, if you have a create routine declared for the class, the create routine must be called. In the examples it is the *make* routine. In order to flag a routine as a create routine, you must include it in the **create** clause at the start of a class. (This clause used to be **creation**, but now the simple word **create** is used, so you don't have to remember the difference between **creation** and **create**.) Create routines are a bit like constructors in C++ and Java. More than one can be declared, but unlike C++ and Java constructors, you can declare several create routines with the same signature.

```
class
  CAR
inherit
  VEHICLE
create
  make
feature
  make is -- a create routine
  do
    ...
end
end -- CAR
```

Also different to C++ and Java is the fact that create routines can be called as normal routines. That is so long as the creation routines are exported as normal routines. The export status as a create routine and a normal routine can be different, so, if you really don't want your create routines called as normal routines, you can prevent this.

Note that the **!!** creation syntax is still supported by Eiffel for CodeWarrior. You will get a warning if the parser finds it in any .e file. You should not use the **!!** form anymore—**create** does everything.

Eiffel has no delete instruction. This is because, as with Java, Eiffel is garbage collected. Garbage collection is known to completely cure the programming ills of dangling pointers and memory leaks. This greatly simplifies the programming effort by removing one of the largest bookkeeping headaches for programmers. Garbage collection has also proven to be very efficient in modern implementations.

Other Instructions

In order to write routines, you use a sequence of *instructions*. As a point of terminology, Eiffel calls these instructions rather than statements, as in other languages. Eiffel provides the usual kinds of instructions: routine call, assignment, if then elseif ... else, loop, object creation and inspect (case or switch). These are just about the only instructions that Eiffel provides. Much of the power of Eiffel is provided in the libraries, which build upon the basic features of the language.

Class variables

If you have used Smalltalk, C++, or Java, you will be wondering how to create class variables; that is variables which do not have one copy per object, but one per class of objects. The Eiffel equivalent for doing this is **once** routines. These are covered by the many other tutorials and books on Eiffel.

Design by Contract

One of the most significant aspects of Eiffel is that it is not only a language with which you can write executable software, but it is a language that embodies many design concepts. Thus you can use the same language for design and implementation. The notion of design by contract is a formal way to divide up the work between a routine and its caller, so that all the work is done and it is not repeated, which would cause performance problems.

Not only is design by contract a formal way of designing interfaces, but a good way to document interfaces. Often routine signatures are not enough to show how to call a routine—contract information adds the extra details.

The most noticeable language features are the pre and postconditions of routines in the *requires* and *ensures* clauses. An example is:

```
square_root (n: REAL): REAL is
```

```
    require
```

```
        n >= 0
```

```
    do
```

```
    Result := ... sqrt calculation
ensure
    n = Result * Result
end -- square_root
```

Here the `requires` clause tells the caller that they are responsible for checking that the argument passed is nonnegative. The `ensure` clause tells us some properties of the calculation and the conditions we expect to hold after the routine is complete. Not only do these clauses document what is expected of the caller and of the routine itself, but these are checked at run time to make sure the software is operating correctly (as long as assertion monitoring is turned on—it can be turned off once you are confident that the software is working correctly).

If the assertion checks fail, an exception is raised. If the `require` clause fails, an exception is raised in the caller. If an `ensure` clause fails, an exception is raised in the routine itself. Exceptions may be caught with a `rescue` clause and if able to be corrected, the routine can be restarted with the `retry` instruction:

```
square_root (n: REAL): REAL is
  require
    n >= 0
  do
    Result := ... sqrt calculation
  ensure
    n = Result * Result
  rescue
    ... clean up instructions
    ... if cleanup successful...
  retry
end -- square_root
```

If a `retry` instruction is not executed in the body of the `rescue` clause, the routine fails, and an exception is raised in the caller. If the precondition `n >= 0` fails, then the exception is raised in the caller since in design by contract, they have failed their obligation.

Not only can preconditions check parameters, but they can also check the object state to ensure that the object has been set up correctly prior to a routine call. For instance, consider a `WINDOW` object. Before being able to move the `WINDOW`,

the window must be open, so a requirement of the *move* routine would be that the window is open.

Note that this gives a level of documentation that Mac programmers have always wanted—what order should things be done in.

Another aspect of design by contract is the class **invariant**. The class invariant always makes sure that objects are in a valid state. This is closely related to creation routines because creation routines must initialise the state of an object so that the class invariant is satisfied. For normal routines to execute correctly, not only must their requires clause be met, but the class invariant must also be satisfied. A normal routine must also leave an object in a valid state, so the class invariant is always checked when a routine completes. (In fact, it is a little more complicated than this in the case where a chain of routines are called on the same object, but we needn't concern ourselves with that here.)

As documentation a class invariant precisely captures the properties of a class and, therefore, is a very important part of class design.

Eiffel does not have...

Gotos and global variables. Gotos are not needed because the Eiffel style is to write small routines. Global variables are a sign of poor structuring—all Eiffel code must be structured in classes. **Once** routines, which have already been mentioned, are the concept needed to do away with global variables since **once** routines give controlled access to shared information. Eiffel also does not need type casts to make up for a flawed type system and, like Java, does not have pointers with their associated problems.

The Inner (outside) world

In order to interface to existing software or low-level (inner) software, Eiffel provides **external** routines. These too can be guarded with pre- and postconditions. The only difference between a normal routine and an external routine is that the **do** section is replaced by the **external** keyword and some information on how to link to the external software, but has no instructions.

The Cocoa and Carbon libraries on OS X make extensive use of the external feature to interface. Eiffel Carbon has been designed so that OS X Carbon has a object-oriented wrapping and so the Eiffel interface is more organized than just the raw Carbon API.

If you need to write your own C routines either for performance or direct access to hardware (which you normally should not do since you should call the toolbox), you will call these via Eiffel's external routines.

Where to now...

That's about all there is to Eiffel the language. The rest is up to you by using good design of software that Eiffel will help you with, more than any other language, and intelligent use of the Eiffel libraries such as EiffelBase and Cocoa that give you the features that other languages have built in. If you are writing Eiffel on the Macintosh, a very good place to start is to study the Eiffel Carbon library. Carbon shows all the advanced features of Eiffel and how the interface to MacOS can be organized much better than previously possible—such are the possibilities when using Eiffel to design, write and organize your software.

Eiffel for Macintosh OS X ProjectBuilder and CodeWarrior are available from <http://www.maceiffel.com/>.

For implementations of Eiffel on other platforms:

- <http://www.object-tools.com>
- <http://www.eiffel.com>
- <http://www.loria.fr/projects/SmallEiffel>
- <http://www.bon-method.com>

Books

Object-oriented Software Construction, Bertrand Meyer (Prentice Hall).

Eiffel: The Language, Bertrand Meyer (Prentice Hall).

Objects Unencapsulated: Java, Eiffel, and C++??, Ian Joyner (Prentice Hall).

For advice and consulting on Eiffel contact i.joyner@acm.org
