



IEFFEL

PROGRAMMING LANGUAGES ASSIGNMENT 2



APRIL 24, 2014

MASWE L.T: 201101428

MARUMO O.R: 201101426

SEIPEI O: 201102783

Table of Contents

PRIMITIVE DATA TYPES	2
POINTER/REFERENCE TYPES	3
OVERLOADED OPERATORS	4
TYPE CONVERSIONS	5
MULTIPLE-WAY SELECTION.....	6
ITERATION BASED ON DATA STRUCTURES/LOGICALLY-CONTROLLED LOOPS	7
TWO PARAMETER PASSING METHODS FOR SUB-PROGRAMS	9
ABSTRACT DATA TYPES	10
CONCURRENCY.....	12
EVENT HANDLING	14
BIBLIOGRAPHY	15

PRIMITIVE DATA TYPES

Primitive types are predefined types of data, which are supported by a programming language (Christensson, 2009). The primitive types in Eiffel are represented as a class as in Eiffel every object is an instance of a class. These basic types includes but not limited to;

- Boolean
- Character
- Integer
- Integer_64
- Real
- Double
- String

Eiffel is a strongly type language meaning the primitive types are divided into reference types (the value of some primitive types are references to an object, not the object itself) and expanded types (where values are actual objects). For example; the value 3 of type INTEGER is really an object to the type INTEGER with a value 3, not a pointer to an object of type INTEGER. Primitive types in Eiffel are immutable. These primitive types therefore does not allow boxing and unboxing from or to wrapper classes since they are already just normal classes.

INTEGER

An entity of type INTEGER represents an integer value (i.e. an integer object) and not references to an integer object. It is not possible to change the value of an INTEGER, i.e. with a variable *i* of type INTEGER, there is no operation like *i.increment*. The only possibility to change the value of a variable of a basic type is to assign a new value to it, e.g. like *i := i + 1*.

The sized variant can be chosen by a compiler option, with its default being Integer_32. An INTEGER_n represents a signed integer value in the range $-2^{(n-1)} .. 2^{(n-1)} - 1$, where *n* is either 8, 16, 32 or 64. I.e.

```
INTEGER_8:  -128 .. 127
INTEGER_16: -32768 .. 32767
INTEGER_32: -2147483648 .. 2147483647
INTEGER_64: -9223372036854775808 .. 9223372036854775807
```

```
local
  year: INTEGER
do
  ...
  if year \ 4 = 0 and year \ 100 /= 0 or year \ 400 = 0 then
    print ( year.out + " is a leap year%N" )
  else
    print ( year.out + " is a leap year%N" )
  end
  ...
End
```

NATURAL

The NATURALs are unsigned and represent values in the range $0 \dots 2^{(n-1)}$

```
NATURAL_8:  0 .. 255
NATURAL_16: 0 .. 65535
NATURAL_32: 0 .. 4294967295
NATURAL_64: 0 .. 18446744073709551615
```

BOOLEAN

Only holds values True and False in this case. These are used with Boolean operators;

```
not           -- unary
and or xor    -- binary strict
and then or else implies -- binary semistrict
```

Mostly Adapted from (Meyer B. , Eiffel Power, 2001).

POINTER/REFERENCE TYPES

The default type of objects declared in Eiffel is references (essentially pointers without the associated dangers). This is accomplished by invoking a creation routine on the new object, as the following example demonstrates: (Bartrand, 1990)

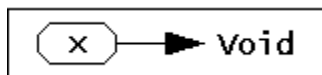
```
x : SOMECLASS;
!!x.make;      -- call the a SOMECLASS creation routine
x.some_routine; --some_routine is a member of SOMECLASS
```

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.

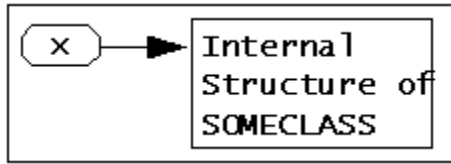
Before !!x.make was called, x looked like this:



1.

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:



2. A reference variable may be attached to an object at run-time; thus, it is effectively a (safe) pointer to a memory location. The pointer is "safe" in the sense that its address cannot be taken (easily), and pointer arithmetic cannot be applied to it. To use a reference variable requires two steps: declaration of the variable, and allocation/attachment of an object to the variable.

OVERLOADED OPERATORS

Overloading operators defines the multiple use of an operator where one operator (say "+") can be used for addition of integers, floats and can also be used to concatenate strings in some programming languages (Sebastia, 2012).

Operation overloading in Eiffel is mostly compared to Ad-hoc polymorphism simply because Ad-hoc polymorphism is different operations on different types known by the same name. Eiffel has some exceptions that it supports under overloading operators to optimize its performance;

1. All operations in the system must be messages to objects
 - Executions with operators are considered to be equivalent to message calls
2. Operators must have equivalent functional forms `1 + 2` is the same as `1. + (2)`

Eiffel does not have any operator overloading and this disadvantaged the language because it failed to take advantage of user's existing knowledge of the operator, failed to capture the readability that can be facilitated by overloading an operator by succinct the operator.

To overcome these disadvantages, Eiffel managed to use a built in keyword "alias" where to overload "+" for example for using it to add integers, floats, real. Eiffel achieves the same effect by a combination of overriding (known in Eiffel as "redefining"), aliases and conversions.

For example if you want to have several versions of "+":

a + b for a, b: INTEGER

a + b for a, b: REAL

a + b for a, b: VECTOR [INTEGER]

then in Eiffel it simply means that the three classes involved each have a feature

plus alias "+" (other: CLASS): CLASS

where *CLASS* is the given class (*INTEGER*, *REAL*, *VECTOR [G]*)

Specific declared example:

```

class INTEGER feature
  ...
  plus alias "+" (other: like Current): like Current
  do ... end

  plus product "*" (other: like Current): like Current
  do ... end
  ...
end

```

Thus even though Eiffel doesn't have operator overloading, it doesn't have the disadvantages faced by other programming languages without overloading operators (Meyer B. , Basic Eiffel language mechanisms, 2006).

TYPE CONVERSIONS

In most object oriented languages, type conversions are easily related to the concept of inheritance which has proven to be a very useful tool to these languages (Meyer, 2006). A pullout quote by Bertrand Meyer stated that a type may conform or convert to another, but not both. Eiffel provides a numerous number of mechanisms in automatic type conversions. The goal is to be able to convert from a basic primitive types such that add for example a complex number and an integer, in order to accomplish this a two-sided conversion is built into Eiffel programming language. It is possible to convert from and to convert to a particular data type using a convert clause. The mechanism coexists with inheritance and complements it. To avoid any confusion between the two mechanisms, the design enforces the following principle:

(Conversion principle) A type may not both conform and convert to another

Eiffel uses both implicit (automatic) and explicit (user defined) conversion; Where in implicit conversion it mostly uses implicit coercion (which is mostly known as type narrowing) by object testing such that from a sub to super-type and by obeying ad-hoc rules.

Example

```
your_real := 10
```

Which is a short hand for **create** your_real.from_integer (10)? A real type has been assign an integer type and Eiffel allows such that the 10 value is automatically converted to a real during execution to prevent an error from occurring since Eiffel is strongly typed (Errors easily caught). It assists the user not since errors will not be raised and the user need not worry about intense learning of the language (amateurs may easily use it).

In Eiffel explicit conversion where a user specifically and unambiguously states that they want a specific type to be converted to another type. Under this sort of conversion Eiffel mostly uses

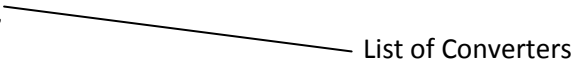
explicit coercion and down casting and upcasting. In explicit coercion the user uses the keyword “**create**”, to specify the object conformed from and conformed to.

For example **create** your_real.from_integer (10), a value 10 of type integer is been converted from integer to a real number. This sort of conversion allows for late binding of values and variables which is has been proven to advantageous to the programming language (Sebasta, 2012).

In down casting the reference of a base class to one of its derived classes such that from a base class to its super class while up casting is the other way round. A built in keyword “**convert**” is used to do the actual conversion.

For example

```
expanded class REAL_64 inherit ... create
    from_integer, from_real_32, ...
convert
    from_integer ({INTEGER}),
    from_real_32 ({REAL_32})
feature -- Initialization
    from_integer (n: INTEGER)
        -- Initialize by converting from n.
        do
            ... Conversion algorithm
        end
    ...
End
```



The advantages of this sort of conversion are that it uses both the reference and expanded types that are defined in Eiffel. It allows for both polymorphism (which Polymorphism is used to define flexible entities that may become attached to objects of various forms at run time (Karine Arnout(Axa Rosenberg, 2006) and inheritance.

MULTIPLE-WAY SELECTION

These statements allow selection of one of any number of statements or statement groups (Sebasta, 2012). The following is the syntax for Eiffel multiple way selection, adapted (Eiffel):

```
inspect val
    when v1 then statement1
    when v2, v3 => statement23
    else statement_else
end
```

here val is the control expression which can either be an integer or a character and v1 and v2 are the selectable statements. The else clause specifies the default statement if the selectable

statements all evaluate to false. If the programmer does not include the else, which is optional, an exception is triggered.

An example of using inspect to see what day of the week it is (Gibbons & Dúnlaing), i is declared an integer and the result a string. Here i is the integer control expression.

```
weekday (i : INTEGER) : STRING is
do
  inspect i
  when 0 then Result := "Sunday"
  when 1 then Result := "Monday"
  when 2 then Result := "Tuesday"
  when 3 then Result := "Wednesday"
  when 4 then Result := "Thursday"
  when 5 then Result := "Friday"
  when 6 then Result := "Saturday"
end
end
```

The selectable statements in Eiffel are statement sequences

ITERATION BASED ON DATA STRUCTURES/LOGICALLY-CONTROLLED LOOPS

LOGICALLY CONTROLLED LOOPS

The repeated execution of a collection of statements is based on a Boolean expression (Sebesta, 2012), that is the expression evaluates to either true or false. In Eiffel, loops adhere to the following syntax (Eiffel Syntax Summary):

```
from stmt
  until cond
  loop stmt
end
```

The until clause is the one that determines whether the loop body will be executed or not. An example of a logically controlled loop (EiffelSoftware, 2014)

```
from
  my_list.start
until
  my_list.off
loop
```



```

        print (my_list.item)
    my_list.forth
end

```

However, there is another way to accomplish the above

```

across my_list as ic loop print (ic.item) end

```

Loops in Eiffel are pretest; the loop body is executed only after the expression has been tested/evaluated. The logically controlled loops are a special case of the counting loop statement. The loop is basically a combination of both counter- and logically-controlled. The following example illustrates such, where the until clause has an expression should evaluate to true for the loop to end and inside the loop, the control variable is incremented.

```

from
    i := 1
until
    i = 10
loop
    io.put_int (i);
    io.new_line;
    i := i + 1;
end
(Fox, 2005)

```

This type of looping is helpful as it has writability. The programmer is able to express themselves and there is less to learn in a language.

ITERATION BASED ON DATA STRUCTURES

Eiffel has inbuilt functions in its library that assist in data structure iteration. However, iteration can be possible with external iterators, not available to the library. The downside is, the programmer has to keep track of the traversal state. For example

```

array_iteration(my_array: ARRAY[CHARACTER]) is
local
    i: INTEGER; element: CHARACTER;
do
    from i := my_array.lower; until i > my_array.upper
    loop
        element := my_array.item(i);
        -- Iteration body itself.
    end
end

```

```

        i := i + 1;
    end;
end;

```

Iteration is different for all data structures in this case. With the inbuilt function from the library the iteration becomes easier as the ITERATOR itself keeps track of the traversal state (Zendra & Colnet). Example

```

iterator_scheme(my_iterator: ITERATOR[CHARACTER]) is
local
i: INTEGER; element: CHARACTER;
do
from my_iterator.start; until my_iterator.is_off
loop
element := my_iterator.item
-- Iteration body itself.
my_iterator.next
end
end

```

TWO PARAMETER PASSING METHODS FOR SUB-PROGRAMS

Within the Eiffel software development method and language, the terms argument and parameter have distinct uses established by convention. The term argument is used exclusively in reference to a routine's inputs, and the term parameter is used exclusively in type parameterization for generic classes (Meyer, 2006)

Eiffel uses two types of parameter passing

PASS BY VALUE

To make a routine general purpose it has *parameters*, these are sort of dummy variables that are set when the routine is called. Each call on the routine can provide different values (*actual parameters*) for the routine to work with, we could for example call larger like this:

```
io.put_integer( larger(x, y) )
```

the actual parameter values are taken from x and y. In the heading of the routine the names and types of the parameters are defined in parentheses (round brackets) after the routine name. The parameter names defined and used in the routine are sometimes called *formal parameters*

to differentiate them from the actual parameter values provided in the call. If there are two or more parameters of the same type they can be defined separated by commas:

larger(a,b: INTEGER): INTEGER is...

if there are parameters of different types the definitions are separated by semi-colons:

cyril(n: INTEGER; x, y: DOUBLE)

routine cyril is a procedure that takes three parameters, an integer and two doubles.

Note that the way actual values are associated and assigned to the is by position not name. The names of the formal parameters can be anything and need not match any names in the calling code.(Bertrand, 1997)

PASSBYRESULT

A procedure has no result but a function does and its type must be specified after the parameter list (if any). So:

larger(a,b: INTEGER): INTEGER is

...

for the larger function the result type is INTEGER.

When the function is called:

largest := larger(largest, num)

the actual data being passed to the function are copied to the corresponding parameters (first argument to first parameter, second one to second parameter). Then the processor “jumps” to the first statement of then function. The routine executes to the end when it returns, if it's a function the processor returns to continue evaluation of the expression it was called from, if is a procedure the processor returns to the statement after the call. In order for a function to return a result it is necessary for the function to assign to the special name result, this is the value that will be carried back to the expression when it returns.

ABSTRACT DATA TYPES

A class is a representation of an Abstract Data Type (ADT). Every **object** is an instance of a class. The object creation uses a so-called **creation procedure**. Eiffel further distinguishes between routines that return a result (**functions**) and routines that do not return a result (**procedures**) . This is a classification by implementation : routines vs. attributes, namely computation vs. memory, Eiffel distinguishes between “commands” and “queries”. Even if not enforced by any compiler, the Eiffel method strongly encourages following the **Command/Query Separation principle**: A feature should not both change the object’s state and return a result about this object. In other words, a function should be side-effect-free. (Bertrand, 1997)

Another important principle, which is **Information Hiding**: The supplier of a module (typically a class) must select the subset of the module’s properties that will be available officially to its

client (the “public part”); the remaining properties build the “secret part”. The Eiffel language provides the ability to enforce this principle by allowing to define fine-grained levels of availability of a class to its clients.

Another principle enforced by the Eiffel method and language is the principle of **Uniform Access**, which says that all features offered by a class should be available through a uniform notation, which does not betray whether features are implemented through storage (attributes) or through computation (routines). Indeed, in Eiffel, one cannot know when writing `x.f` whether `f` is a routine or an attribute; the syntax is the same. (Bartrand, 1990)

note

description: Objects that model lists

revision: \$Revision:1.4 \$

class

OLD_FASHIONED_LIST [G]

obsolete "This class is obsolete, use LINKED_LIST [G] instead"

inherit

DYNAMIC_LIST[G]

create

make

feature-- *Initialization*

make

-- *Create an empty list.*

do

before := True

ensure

is_before: before

end

feature-- *Access*

item: G

-- *Current item*

do

Result := active.item

end

first: **like** item

-- *Item at first position*

do

Result := first_element.item

end

... other features omitted ...

invariant

before_constraint: before **implies**(active =first_element)

after_constraint: after **implies**(active =last_element)

CONCURRENCY

Concurrent programming is considered to be a challenging and inherently error prone process with of some difficulty in the implementation process of a powerful concurrent programming tool or language. To incorporate this into Eiffel, SCOOP (Simple Concurrent Object Oriented Programming) is built into a standard Eiffel (Meyer, 1997). SCOOP is a simple but yet powerful notation that allows for concurrency by extending coverage of fully-fledged concurrency and distribution constructs but in minimal terms. SCOOP covers important terms in concurrency like producer-consumer, synchronized objects, distributed objects and threads. The combination of SCOOP and Eiffel provides good motivation for developing actual executable target code.

In general explanation, the concurrency constructs of SCOOP extend the language by adding a keyword “separate” which will be applied to attributes, classes and formal routine arguments. The application of this keyword to either a class, an attribute and/or formal routine argument will allow that particular class or its equivalent to execute in their own thread of control or the application of separate to routine arguments indicates that these objects are points of synchronizations and can be safely shared among concurrent threads.

Example of the separate argument rule

The base class uses a feature launch_producer (and a corresponding feature launch_consumer) for instructing the producers and consumers on how many products to handle. launch_producer looks like this:

```
launch_producer (a_producer: separate PRODUCER)
-- Launch `a_producer'.
do
  a_producer.produce (900)
end
```

It might occur to you that it would be easier, simpler, and clearer just to include this feature's single procedural line:

```
a_producer.produce (900)
```

The object attached to a_producer is declared of a separate type), and according to the separate argument rule, calls on a separate object are valid only when applied to an argument of the enclosing routine.

Two processes share a fixed-size buffer. One process produces information and puts it in the buffer, while the other process consumes information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently (Boer, 2004).

In order to safely synchronize these processes, we (a) use some mechanism to provide mutual exclusion so that only one process at a time can access the buffer (otherwise the information in the buffer might be garbled), and (b) we must block the producer when the buffer is full, and block the consumer when the buffer is empty (Fuks, 2004).

The SCOOP version of the producer-consumer provides the same behavior as the Java solution (using wait (), notifyAll (), sleep ()), but with the simplification that only one extra keyword separate is used (instead of Thread, synchronize and wait/notify).

RACE CONDITION

It falls under the competition synchronization where tasks or processes require access to the most important shared resources (CPU, memory etc.) at the same time.

1. Avoid deadlocks: make certain that no two executing threads of control wait perpetually because each needs a resource which is under the control of the other.
2. Ensure fairness: make certain that every participating thread of control eventually gets the opportunity to execute.

Eiffel usually save the race condition problem by using a counter on these resources, by demonstrating mutual exclusion on shared resources where only one process can access that resource at a time. It uses a Mutex class to achieve the above mentioned (such to indicate whether a resource is free to be used) (Eiffel, 2014).

```
my_mutex: MUTEX -- Declaration of the mutex
create my_mutex.make-- Creation of mutex
my_mutex.lock -- Locking the mutex
my_mutex.unlock-- Unlocking the mutex
my_mutex.try_lock --try_lock: if it is not locked yet, lock the mutex and return True, otherwise
it returns False.
my_mutex.is_set-- Is my mutex initialized?
my_cond: Condition Variable

create my_cond.make--Wait for a signal (send by signal). You need to use a mutex.
my_mutex: MUTEX
...
create my_mutex.make
my_mutex must be locked by the calling thread so as wait can be called. wait atomically unlocks
my_mutex and waits for the condition variable my_cond to receive a signal. As soon as it
received a signal, my_cond locks my_mutex
my_mutex.lock-- You must lock 'my_mutex' before calling wait.
```

`my_cond.wait (my_mutex)--` Here the critical code to execute when ``my_cond'` received a signal.

`my_mutex.unlock--` Unlock the mutex at the end of the critical section.

EVENT HANDLING

An event is an external action that happens during program execution. Event-driven programming languages are mostly used for Graphical User Interfaces (GUIs). An example of a simple event handling:

(if "The latest event was `left mouse click on button 23'" then "Appropriate instructions" else if ...), which takes the form (control, event, operation).

These events in Eiffel are handled by agents, defined as objects that represent a routine, which are created based on procedures called for certain events (EiffelSoftware, I2E: Event-Driven Programming and Agents, 2014). They address the control, event, operation issue.

An agent is created as:

`agent r`

where `r` is the routine and used as follows

`ok_button.select_actions.extend (agent your_routine)`

which says: "add `your_routine` to the list of operations to be performed whenever a select event (left click) happens on `ok_button`". `ok_button.select_actions` is the list of agents associated with the button and the event; in list classes, procedure `extend` adds an item at the end of a list (Meyer, 2001).

Eiffel has a library for widgets that provide actions for certain events that can be used by agents. Some include:

Action sequences

`file_drop_actions: EV_LITE_ACTION_SEQUENCE [TUPLE [LIST [STRING_32]]]`

`focus_in_actions: EV_NOTIFY_ACTION_SEQUENCE`

`focus_out_actions: EV_NOTIFY_ACTION_SEQUENCE`

`key_press_actions: EV_KEY_ACTION_SEQUENCE`

more

from

https://docs.eiffel.com/static/libraries/vision2/ev_widget_action_sequences_chart.html

BIBLIOGRAPHY

- Bartrand, M. (1990). *Introduction to the Theory of Programming Languages*. Prentice-Hall International Series in Computer Science.
- Bertrand, M. (1997). *Object-Oriented Software Construction*. Prentice Hall.
- Boer, B. d. (2004). *Object-Oriented Language: Eiffel*. ISE.
- Christensson, P. (2009, August 8). *The PC Help Center*. Retrieved from PC.net:
pc.net/helpcenter/answers/primitive_and_non_primitive_data
- Eiffel*. (n.d.). Retrieved from Pixel's home page: <http://rigaux.org/language-study/syntax-across-languages-per-language/Eiffel.html>
- Eiffel Syntax Summary*. (n.d.). Retrieved from <http://www.infor.uva.es/~felix/priii/sintaxis.html>
- Eiffel, S. (2014). *Eiffel Documentation*. Retrieved from Thread library overview:
https://docs.eiffel.com/book/solutions/thread-library-overview#The_class_MUTEX
- EiffelSoftware. (2014). *ET: Instructions*. Retrieved from Eiffel Documentation:
https://docs.eiffel.com/book/method/et-instructions#The_iteration_form_as_a_boolean_expression
- EiffelSoftware. (2014). *I2E: Event-Driven Programming and Agents*. Retrieved from Eiffel Documentation: <https://docs.eiffel.com/book/method/i2e-event-driven-programming-and-agents>
- Fox, D. R. (2005). *Eiffel Programming Language*.
- Fuks, O. (2004). *SIMPLE CONCURRENT OBJECT – ORIENTED: A GENERATOR BASED IMPLEMENTATION*. Toronto.
- Gibbons, H., & Dúnlaing, C. Ó. (n.d.). *SmartEiffel: a short course*. Retrieved from University of Dublin, School of Mathematics:
http://www.maths.tcd.ie/~odunlain/eiffel/eiffel_course/shortcrs/html/notes.html
- Meyer, B. (1997). *Object-Oriented Software Construction, Second Edition*. Prentice Hall.
- Meyer, B. (2001). *Invitation to Eiffel*. Interactive Software Engineering Inc.
- Meyer, B. (2006, August 1). *Basic Eiffel language mechanisms*. Retrieved from Basic Eiffel language mechanisms: <http://se.ethz.ch/~meyer/publications/online/eiffel/basic.html>
- Meyer, B. (2006). *Basic Eiffel Language Mechanisms*. Prentice Hall.
- Sebesta, R. W. (2012). *Concepts of Programming Languages*. Upper Saddle River: Pearson.

Zendra, O., & Colnet, D. (n.d.). *Adding external iterators to an existing Eiffel class library*. Vandœuvre-lès-Nancy Cedex: University Henri Poincaré.