Java & Eiffel: An objective personal assessment

Bertrand Meyer
Chair of Software Engineering, ETH Zurich
Topics

1. Background
2. Common elements
3. Contracts
4. Type system
5. Inheritance
6. Agents
7. Other mechanisms
8. Syntax, ease of learning; conclusion
Background: Eiffel

Eiffel 1: 1986 (contracts, multiple inheritance, genericity, deferred classes…)

Eiffel 2: 1988 (exceptions, constrained genericity)

Eiffel 3: 1991 (uniform type system, infix/prefix features, …)

1997: Agents, Precursor

2005-2006: ECMA/ISO standard: attached types, numerous clarifications and simplifications

2008-now: Void safety, concurrency (SCOOP)

In progress: advanced functional features, safe covariance
Background: Java

1995: 1.0
1997: 1.1
   Microsoft JVM, Swing
1999: 1.2 (Java 2)
   Java Foundation Classes
2000: 1.3
   Performance improvements, Hotspot
2004: 1.5 (5.0):
   Metadata, genericity
2006: Java SE 6, support for scripting languages
2011: Java SE 7, support for dynamic languages
2014 (expected): Java SE 8, lambda expressions
C# - Originally 1999 (COOL), part of .NET
- 2002: C# 1.0
- 2006: C# 2.0, generics, partial types
- 2007: C# 3.0, extension methods, lambda expressions
- 2010: C# 4.0, generic co- and contravariance
- 2012: C# 5.0, asynchronous methods
What’s common

Not C++
Not backward-compatible with C (but Java closer to C, especially syntax)
Object-oriented languages
Statically typed languages
Dynamic binding by default
Type system permits garbage collection
Genericity (built-in in Eiffel, late addition in Java)

Portable implementations
Overall structure

Java: classes, but also static methods

Eiffel: classes throughout - unit of both type and module decomposition
The problem with attribute export status

If an attribute is exported, clients can both read it and assign any value that they want to it.

Ex: heater.temperature = 19;
Information hiding

In Java:

Can still do \( x.a := v \)

This design mistake (in my opinion) comes from C++: designers did not understand the Uniform Access principle.

Exporting an attribute means exporting it read-write.

Eiffel approach (Uniform Access):
- Query can be attribute or function.
- Client does not know which - only that it’s a query.
- (Difference not visible in “contract view” of class).
- Exporting a query means exporting it to read; there’s nothing wrong or dangerous with this.
- To provide setter privileges: write procedure.
- Can use assignment-like syntax for setter.
Consistency principle

The language should offer one good way to do anything useful
Language style

Compatibility principle

Traditional notations should be supported with an O-O semantics
Infix and prefix operators

In

\[ a - b \]

does not make sense.

the \(-\) operator is "infix"
(written between operands)

In

\[ -b \]

the \(-\) operator is "prefix"
(written before the operand)
The object-oriented form of call

\texttt{some\_target.some\_feature(some\_arguments)}

For example:

\texttt{my\_figure.display}

\texttt{my\_figure.move(3, 5)}

\texttt{x := a.plus(b)}
Operator features

expanded class INTEGER feature

+ alias "+" (other: INTEGER): INTEGER
  -- Sum with other
  do ... end

times alias "*" (other: INTEGER): INTEGER
  -- Product by other
  do ... end

- alias "-": INTEGER
  -- Unary minus
  do ... end

... end

Calls such as \textit{i.plus(j)} can now be written \textit{i + j}
If class \( A \) has an attribute \( \text{att} : \text{SOME\_TYPE} \), what may a client class \( C \) with \( a : A \) do with \( a.\text{att} \)?

The attribute may be:

- **Secret**
- **Read-only**
- **Read, restricted write**
- **Full write**

- \( a.\text{att} \) invalid
- \( a.\text{att} \) permitted in \( C \) (for access)

Modify through:
- \( a.\text{some\_procedure} \)
- \( a.\text{set\_att}(v) \)
Abstraction and client privileges

If class \( A \) has an attribute \( \text{att} : \text{SOME\_TYPE} \), what may a client class \( C \) with 

\[
a : A
\]
do with \( a.\text{att} \)?

**Read access if attribute is exported**

- \( a.\text{att} \) is an expression.
- An assignment \( a.\text{att} := v \) would be syntactically illegal!

(It would assign to an expression, like \( x + y := v \).)
Applying abstraction principles

Beyond read access: full or restricted write, through exported procedures.

Full write privileges: `set_attribute` procedure, e.g.

```
set_temperature (u : REAL) is
    -- Set temperature value to u.
    do
        temperature := u
        ensure
            temperature = u
    end
```

Client will use e.g. `x.set_temperature (21.5)`
Other uses of a setter procedure

\texttt{set\_temperature (u : REAL) is}
  \begin{quote}
  \texttt{-- Set temperature value to u.}
  \end{quote}
\texttt{require}
  \begin{quote}
  \texttt{not\_under\_minimum: u >= -273}
  \texttt{not\_above\_maximum: u <= 2000}
  \end{quote}
\texttt{do}
  \begin{quote}
  \texttt{temperature := u}
  \texttt{update\_database}
  \end{quote}
\texttt{ensure}
  \begin{quote}
  \texttt{temperature\_set: temperature = u}
  \end{quote}
\texttt{end}
Having it both ways: assigner commands

Make it possible to call a setter procedure

\texttt{temperature: REAL assign set\_temperature}

Then the syntax

\texttt{x.temperature := 21.5}

is accepted as a shorthand for \texttt{x.set\_temperature (21.5)}

Retains contracts etc.
Eiffel: providing an assigner command

class \textit{C}[G] \text{ feature}

\textit{item} : G

\textit{put} (x : G)

\text{require}

\ldots

\text{do}

\textit{item} := x

\text{ensure}

\textit{item} = x

\text{end}

\text{end}

\textbf{Client code:}

\textit{n} : \text{INTEGER}

\textit{x} : \textit{C}[\text{INTEGER}]
Information hiding

class A

feature
  f ...
  g ...

feature {NONE}
  h, i ...

feature {B, C}
  j, k, l ...

feature {A, B, C}
  m, n...

end

Status of calls in a client with a1:

- a1.f, a1.g: valid in any client
- a1.h: invalid everywhere (including in A's own text!)
- a1.j: valid only in B, C and their descendants (not valid in A!)
- a1.m: valid in B, C and their descendants, as well as in A and its descendants
An example of selective export

**LINKABLE** exports its features to **LINKED_LIST**

- Does not export them to the rest of the world
- Clients of **LINKED_LIST** don’t need to know about **LINKABLE** cells.
Exporting selectively

class LINKABLE [G]
feature {LINKED_LIST}

put_right(...) is do ... end

right: G is do ... end

... end
Information hiding

Information hiding only applies to use by clients, using dot notation or infix notation, as with \( a1.f \) (Qualified calls).

*Unqualified* calls (within class) not subject to information hiding:

```plaintext
class A feature {NONE}
    h is ... do ... end
feature
    f is
do
    ...; h; ...
end
end
```
Possible client privileges in Java

Access specifiers (placed in front of each definition for each member of the class):

- public
- protected
- Package access (no keyword)
- private
Access specifiers

**public**
- The member declared to be public is available to everyone

**private**
- No one can access that member except the class that contains that member, inside methods of that class

**protected**
- Member can be accessed by
  - Descendants of the class
  - Classes in the same package

**Package access**
- Default
- Also called “friendly”
- All other classes in current package have access to that member
- To all classes outside of current package, the member appears to be private
Access modifiers at the class level

Either public or default (no access modifier)

- **public**
  - Appears before the class keyword
  - Makes the class available to a client programmer

- No access modifier
  - Makes the class available only within the package

No private and protected!
## Comparison: Eiffel vs. Java

<table>
<thead>
<tr>
<th>Access level</th>
<th>Eiffel</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>only current class</td>
<td>-</td>
<td>private</td>
</tr>
<tr>
<td>only current class and its descendants</td>
<td><code>feature {NONE}</code></td>
<td>-</td>
</tr>
<tr>
<td>current class + &quot;friends&quot;</td>
<td><code>feature {B, C}</code>(&quot;friends&quot; = B, C and their descendants)</td>
<td>default (&quot;friends&quot; = classes in the same package)</td>
</tr>
<tr>
<td>current class + its descendants + &quot;friends&quot;</td>
<td><code>feature {A, B, C}</code>(&quot;friends&quot; = B, C and their descendants, A = current class)</td>
<td>protected (&quot;friends&quot; = classes in the same package)</td>
</tr>
<tr>
<td>everyone</td>
<td><code>feature {ANY}</code></td>
<td>public</td>
</tr>
</tbody>
</table>
Eiffel - no package mechanism
Eiffel - no way of hiding a feature from your descendants

- Module viewpoint: If B inherits from A, all the services of A are available in B (possibly with a different implementation).

Java - no way of exporting a member only to self and descendants
Java - no language rule to distinguish between access to attributes for reading and for writing
Java - additional way of making a class available outside its package or not

Access control more fine grained in Eiffel
C# access modifiers

C# adds the internal access modifier, which restricts access within the assembly.

Classes can be:
- public
- internal

Class members can be:
- public – accessible to everyone
- internal – accessible only from current assembly
- protected – accessible only from containing class or types derived from containing class (a.k.a. “family” export status)
- protected internal – accessible only from current assembly or types derived from the containing class
- private – accessible only from containing type
The problem with attribute export status

If an attribute is exported, clients can both read it and assign any value that they want to it.

Ex: `heater.temperature := 19`
The C# solution: properties

```csharp
public class Heater {
    private int TemperatureInternal;

    public int Temperature {
        get { return TemperatureInternal; }
        set {
            if (!InRange(value)) {
                throw new ArgumentException("Temperature out of range");
            }
            Temperature Internal = value;
            NotifyObservers();
        }
    }
}
```
Assignment commands

It is possible to define a query as

\[
\text{temperature: REAL assign set_temperature}
\]

Then the syntax

\[
x.\text{temperature} := 21.5
\]

is accepted as an abbreviation for

\[
x.\text{set_temperature} (21.5)
\]

Retains \textbf{contracts} and any other supplementary operations
Contracts

Elements of specification associated with the code
Help in: analysis, design, debugging, testing, maintenance, management

Eiffel: Built-in

Java: additions (iContract, JML)
public /*@ pure @*/ strictfp class Polar extends ComplexOps
    /** The angle of this number. */
    private double ang;
    /** Initialize this polar coordinate number ... */
   /*@ requires mag >= 0 && Double.NEGATIVE_INFINITY < ang
    @ && ang < Double.POSITIVE_INFINITY;
    @ ensures this.magnitude() == mag;
    @ ensures this.angle() == standardizeAngle(ang);
    @ also @ requires mag < 0 && Double.NEGATIVE_INFINITY < ang
    @ && ang < Double.POSITIVE_INFINITY;
    @ ensures this.magnitude() == - mag;
    @ ensures this.angle() == standardizeAngle(ang+StrictMath.PI);
    @ also @ requires Double.isNaN(mag) || Double.isNaN(ang)
    @ || Double.NEGATIVE_INFINITY == ang
    @ || ang == Double.POSITIVE_INFINITY;
    @ signals_only IllegalArgumentException; */
Type system

Java:
- “Primitive” types are special, e.g. int, bool, float
- Special class types: Integer, Float, ...

Eiffel: every type is based on a class

  e.g. INTEGER, REAL, BOOLEAN
Conversions in Java

Built-in conversions between primitive types

For reference types: type narrowing (equivalent of object test)
What is the difference between the following (in Eiffel syntax)?

- `my_polygon := my_rectangle`
- `my_real := my_integer`
Conversion

Try to avoid having a special rule for e.g.

\[ 3 + 5.0 \]
Can we generalize conversion?

Without conversion: we exchange strings with .NET as

- **create** `my_string.from_dotnet (her_dotnet_string)`
- `dotnet_routine ("ABCDE").to_dotnet`

With conversions: convert to out-of-control type:

- `my_string := her_dotnet_string`
- `dotnet_routine ("ABCDE")`
Basic type hierarchy
Resolution

Introduce explicit conversion mechanism

As in rest of language, governs all forms of "attachment" (assignment or argument passing)
First change

class STRING create
    make, from_dotnet

convert
    from_dotnet ({{DOTNET_STRING}})

feature
    from_dotnet (s: DOTNET_STRING)
    do
        ...
    end
Second change

class STRING create
    make, from_dotnet

convert
    from_dotnet ({DOTNET_STRING})
    to_dotnet: {DOTNET_STRING}

feature
    to_dotnet: DOTNET_STRING
    do
        ...
    end
Can we generalize conversion?

Without conversion: we exchange strings with .NET through

- `create my_string.from_dotnet(her_dotnet_string)`
- `dotnet_routine(('ABCDE').to_dotnet)`

With conversions:

- `my_string := her_dotnet_string`
- `dotnet_routine("ABCDE")`

Now: abbreviation for this
First change

class STRING create
    make, from_dotnet

convert
    from_dotnet ({DOTNET_STRING})

feature
    from_dotnet (s: DOTNET_STRING)
        do
            ...
        end
The other way around?

Without conversion: we exchange strings with .NET through

- create `my_string • from_dotnet (her_dotnet_string)`
- `dotnet Routine ("ABCDE") • to_dotnet`

With conversions:

- `my_string := her_dotnet_string`
- `dotnet Routine ("ABCDE")`

Now: abbreviation for this
Second change

class STRING create

make, from_dotnet

convert

from_dotnet ({DOTNET_STRING})

to_dotnet: {DOTNET_STRING}

feature

	to_dotnet: DOTNET_STRING

do

... 

end
Conversion principle
No type may both conform and convert to another.

Conversion Non-Transitivity principle
That $V$ converts to $U$ and $U$ to $T$ does not imply that $V$ converts to $T$.

Conversion Asymmetricity principle
No type $T$ may convert to another through both a conversion procedure and a conversion function.
For programmer-defined types

\[ my\_date := [13, " May", 2013] \]
Related change

Allow $5.0 + 3$ and $3 + 5.0$

$5.0 + 3$ is a shortcut for $(5.0).\text{plus}(3)$

But we want $3 + 5.0$ to be a shortcut for

$((3).\text{to_real}).\text{plus}(5.0)!$
Target conversion

In class \texttt{REAL}:

\begin{verbatim}
plus alias "+" convert
do
  ...
end
\end{verbatim}
Control structures

\begin{verbatim}
across my_list as l loop
    op (l.item)
end

require

across emplist as e all e.item.is_full_time end
\end{verbatim}

For \texttt{across} to be applicable, it suffices that the type of the structure inherit from \texttt{ITERABLE}
Multiple inheritance in Eiffel
Multiple inheritance

A class may have two or more parents.

What not to use as an elementary example: `TEACHING_ASSISTANT` inherits from `TEACHER` and `STUDENT`.
Examples of multiple inheritance

Combining separate abstractions:

- Restaurant, train car
- Calculator, watch
- Plane, asset
- Home, vehicle
- Tram, bus
Warning

Forget all you have heard!

Multiple inheritance is **not** the works of the devil
Multiple inheritance is **not** bad for your teeth
(Even though Microsoft Word apparently does not like it:

Object-oriented programming would become a mockery of itself if it had to renounce multiple inheritance.

)
This is **repeated**, not just multiple inheritance

Not the basic case!

(Although it does arise often; why?)
Composite figures
Multiple inheritance: Composite figures

Simple figures

A composite figure
Defining the notion of composite figure

- center
- display
- hide
- rotate
- move

...
In the overall structure

- FIGURE
  - OPEN FIGURE
  - CLOSED FIGURE
    - SEGMENT
    - POLYLINE
    - POLYGON
    - ELLIPSE
    - RECTANGLE
    - TRIANGLE
    - SQUARE

- LIST [FIGURE]

- COMPOSITE FIGURE

- perimeter
- perimeter*
- perimeter**
- diagonal

62
A composite figure as a list

Cursor

item

forth

after
Composite figures

```plaintext
class COMPOSITE FIGURE inherit FIGURE
    LIST[FIGURE]
feature
    display
        -- Display each constituent figure in turn.
        do
            from start until after loop
                item.display
        end
    end
end
... Similarly for move, rotate etc. ...
```

Requires dynamic binding
Going one level of abstraction higher

A simpler form of procedures *display*, *move* etc. can be obtained through the use of iterators.

Use *agents* for that purpose.
Multiple inheritance: Combining abstractions

**COMPARABLE**
- `<`, `<=`, `>`, `>=`, ...
- (total order relation)

**NUMERIC**
- `+`, `-`, `*`, `/`
- (commutative ring)

**INTEGER**

**REAL**

**STRING**

**COMPLEX**
The Java-C# solution

No multiple inheritance for classes

“Interfaces”: specification only (but no contracts)
  ➢ Similar to completely deferred classes (with no effective feature)

A class may inherit from:
  ➢ At most one class
  ➢ Any number of interfaces
Multiple inheritance: Combining abstractions

- COMPARABLE
- STRING
- INTEGER
- REAL
- COMPLEX
- NUMERIC

Relations:
- <, <=
- >, >=
- +, –,
- *, /,

(total order relation)
(commutative ring)
How do we write **COMPARABLE**?

deferred class **COMPARABLE** feature

  *less* alias "<" (x: **COMPARABLE** [G]): BOOLEAN
  deferred
  end

  *less_equal* alias "<=" (x: **COMPARABLE**): BOOLEAN
do Result := (Current < x or (Current = x))
end

greater alias ">" (x: **COMPARABLE**): BOOLEAN
do Result := (x < Current) end

greater_equal alias ">=" (x: **COMPARABLE**): BOOLEAN
do Result := (x <= Current) end
Lessons from this example

Typical example of *program with holes*

We need the full spectrum from fully abstract (fully deferred) to fully implemented classes

Multiple inheritance is there to help us combine abstractions
Non-conforming inheritance

class
  ARRAVED_LIST [G]
inherit
  LIST [G]
inherit {NONE}
  ARRAY [G]

feature
  ... Implement LIST features using ARRAY features
  ...
end
Multiple inheritance: Name clashes

A

C

B

f

f

?
Resolving name clashes

rename f as A_f
Consequences of renaming

\( a_1 : A \)
\( b_1 : B \)
\( c_1 : C \)
...
\( c_1.f \)
\( c_1.A_f \)
\( a_1.f \)
\( b_1.f \)

**Invalid:**
- \( a_1.A_f \)
- \( b_1.A_f \)
Are all name clashes bad?

A name clash must be removed unless it is:

- Under repeated inheritance (i.e. not a real clash)
- Between features of which at most one is effective (i.e. others are deferred)
Another application of renaming

Provide locally better adapted terminology.
Example: *child* (*TREE*); *subwindow* (*WINDOW*)
Overloading

Present in C++, Java, C#, not in Eiffel

Java rule: several features may have the same name if signature (argument types and numbers) are different

Example

print (x: INTEGER)
print (x: INTEGER; f: FORMAT)
print (x: REAL)
Risks of overloading

Conflicts with inheritance, polymorphism, dynamic binding
Causes confusion: what does \( a.f(xx) \) mean?

class POINT feature

... 

  set_cartesian (x: REAL ; y : REAL) do ... end

  set_polar (ro: REAL ; theta: REAL) do ... end 

...

end
Constructors

C++, Java: name of class, overloaded

\[
x = \text{new POINT} (1, 0, "cartesian");
\]

Eiffel: specific creation procedures

\[
\text{create } x.set_cartesian (1, 0)
\]

Can be used as normal procedures: \(x.set_cartesian (1, 0)\)

No special rules for inheritance; each class’s constructors are independent from those of parents.
Exception handling: C++/Java

Try operation and provide alternative mechanism:

```cpp
try {
    instructions_1
} catch (A a1) {
    instructions_A
} catch (B b1) {
    instructions_B
...
finally {
    cleanup
}
```
Raising and specifying exceptions

```java
public static int f (...) throws my_exception
{
    ... throw my_exception
}
```

Then any caller must catch or throw `my_exception`.

But: only for programmer exceptions.
**Agents**

Mechanism to encapsulate operations into objects
Example: Eiffel Event Library

On the publisher side, e.g. GUI library:

- (Once) declare event type:
  
  \[
  \text{click: EVENT\_TYPE [TUPLE [INTEGER, INTEGER]]}
  \]

- (Once) create event type object:

  \[\text{create click}\]

- To trigger one occurrence of the event:

  \[
  \text{click.publish ([x\_coordinate, y\_coordinate])}
  \]

On the subscriber side, e.g. an application:

\[\text{click.subscribe (agent my\_procedure)}\]
Another example of using agents

\[
\begin{align*}
\int_{a}^{b} \text{my\_function} (x) \, dx \\
\int_{a}^{b} \text{your\_function} (x, u, v) \, dx
\end{align*}
\]

my\_integrator\_integral (agent my\_function , a, b) \\
my\_integrator\_integral (agent your\_function ? , u, v), a, b)
No agents ("closures") in Java

Use inner classes

See: java.sun.com/docs/white/delegates.html
Covariance

class DRIVER feature
    transport: VEHICLE
    set_transport (v: VEHICLE) do ... end

end

class TRUCK_DRIVER inherit DRIVER
    redefine transport, set_transport end

feature
    transport: TRUCK
    set_transport (t: TRUCK) do ... end

end
Anchored types

class DRIVER feature
    transport : VEHICLE
    set_transport (v : like transport) do... end
    ...
end

class TRUCK_DRIVER inherit
    COMPANY
    redefine transport end

feature
    transport : TRUCK
    -- No need to redefine set_transport
    ...
end
Type redefinition rule

May redefine argument or result to a descendant of the original type
Covariance pros and cons

Covariance (Eiffel):
   Realistic modeling

But:
   Type checking issues

For that reason Java and many other languages are novariant

This is safer but pushes the problem to the programmer
The problem with covariance

c: COMPANY
bc: BLOOMBERG_COMPANY

v: VALUATION

... c := bc

c.set_valuation (v)

Catcall!

class COMPANY feature
  valuation: VALUATION
  set_valuation (v: like valuation) is
    do ... end
end

class BLOOMBERG_COMPANY inherit
  COMPANY
  redefine valuation end

feature
  valuation: VALUATION
  -- No need to redefine set_valuation
    ... end

Catcalls

Follow from combination of:

- Polymorphism
- Covariant redefinition

\textit{CAT} stands for “Changed Availability or Type”

The attached mechanism of ISO Eiffel removes all catcall possibilities
Once routines

If instead of

\[
\text{If } r \text{ is } \begin{align*}
&\text{do} \\
&\text{end}
\end{align*}
\]

you write

\[
\text{If } r \text{ is } \begin{align*}
&\text{once} \\
&\text{do} \\
&\text{end}
\end{align*}
\]

then *Instructions* will be executed only for the first call by any client during execution. Subsequent calls return immediately.

In the case of a function, subsequent calls return the result computed by the first call.
Implementation

Java: virtual machine

Eiffel: translation to C or .NET virtual machine

Melting Ice Technology
Syntax

Java:
- Symbol-oriented, C-like

Eiffel:
- Basic structures use keywords, lower-level elements use some symbols
- Keyword consistency: simplest applicable word (require, not requires; feature, not features).
Ease of learning

Usenet posting by David Clark, U. Canberra, taught both Eiffel & Java:

*My experience has been that students do not find Java easy to learn. Time and again the language gets in the way of what I want to teach.... The first thing they see is*

```java
public static void main (String [ ] args) throws IOException;
```

*There are about six different concepts in that one line which students are not yet ready to learn....*