

Java and the Object-Oriented Paradigm: Comparison and Evaluation

W. Al-Ahmad and E. Steegmans

Report CW 249, April 1997

Department of Computer Science, K.U.Leuven

ABSTRACT

Java has, undoubtedly, received unprecedented hype. This article examines Java in light of the Object-Oriented (OO for short) paradigm. We investigate whether the hype or part of it is due to the OO features, or due to other features of the language. For that purpose, we compare Java with other major OO languages such as C++, Eiffel and Smalltalk. Furthermore, we evaluate Java to arrive at an unbiased judgement regarding its expressive power of the OO paradigm. We outline some of the new interesting features of Java that we hope will be adopted by the OO community as new characteristics of OO languages. Finally, we present some recommendations that we hope will be considered in new releases of Java.

1. Introduction

As there are nowadays a plethora of Object-Oriented Programming Languages (OOPLs for short), we believe it is imperative and useful to compare Java with other important OOPLs in an unbiased way. In the white paper [1] as well as in other literature, Java is defined as the language that is simple, *object-oriented*, robust, secure, architecture neutral, distributed, threaded, etc. We know that Java has borrowed its syntax from C++ to make it familiar to programmers. But what about the OO concepts of Java, are they borrowed from C++ too? Did Java introduce new (or improved already existing) concepts that can be considered as major shifts in the OO perspective? This article will answer this question objectively and without any prejudices. To do that we compare Java with two well-known and widely used languages C++ and Eiffel. However, where it is appropriate we use other languages such as Smalltalk and Beta. Our evaluation of Java is based on the language features offered by Java and the languages we consider (other ways for languages comparison exist [14]). We will follow the evaluation methodology proposed by Edward Berard [15] in his essay entitled “Evaluating An Object-Oriented Programming Language”. As such we will consider the common, missing, and new features of Java with respect to the other languages. We want to emphasize that this approach does not mean that the language with more features wins, and that with less features loses. We will mainly illustrate similarities and differences between Java and the other languages with respect to the basic mechanisms of the OO paradigm, that are common to and supported by all OO programming languages. These mechanisms include the concepts of object and class, message and method, inheritance and subclass, as well as related issues like encapsulation, polymorphism, and dynamic binding.

This article is organized as follows: section 2, the largest, includes basic mechanisms and central concepts associated with the OO paradigm. In each of its subsections we discuss, compare, and evaluate Java with the other OOPLs with respect to one aspect of the paradigm. Section 2 discusses desirable properties of the OO paradigm such as generic classes and multiple inheritance which are missing in Java. Section 3 discusses additional properties which are becoming increasingly associated with the OO paradigm such as persistency, concurrency, distribution and other issues. Section 5 gives an overall summary of the comparison and the evaluation. Finally, section 6 gives some conclusions.

2. Common Features of OOPLs

Although there is no consensus on the basic characteristics a language must possess to be called OOPL, most OOPLs offer support for the following concepts. We will see that all the languages we consider possess these characteristics, but differ in the way they view and implement them.

2.1. Notion of Object

Any OOPL includes the notion of an object. An object is an abstract entity which understands and executes a set of operations (methods, features) [4]. However, class-based OO programs do not deal with single objects, rather they deal with groups of objects. A class is a description of a group of objects with common characteristics and behavior. Classes and objects are the most important building blocks of the OO paradigm. Java as the other OOPLs fully supports the concept of class and object. In the following subsections we look closely at the language constructs and mechanisms supporting object representation, object behavior, object encapsulation, and object construction and

destruction. We compare the concepts of Java with other OOPs and evaluate the approach adopted by Java.

2.1.1. Object Representation

In principle, each of the characteristics ascribed to the objects of a given class is expressed in terms of a relationship of the object with objects of other classes. There are actually two kinds of relationships between classes :

- The "is-a" relationship - a new class is derived from existing classes due to common characteristics and behavior between these classes. We will look at this type in the section on inheritance.
- The "has-a" relationship - the implementation of a new class uses existing classes. That a class say B uses one or more operations of another class A to implement its behavior. The way the that the behavior of A is implemented is hidden from B.

Object representation thus deals with the second type of relationship. Java uses instance variables (data members in C++ and attributes in Eiffel) to store characteristics of objects of a class. Establishing a proper relationship between objects of different classes leads to a choice between value semantics or reference semantics. In a representation based on value semantics, each object will retain a private copy of an object of the related class. On the other hand using reference semantics implies that each object retains a reference to an object of the related class. Java followed the lines of Eiffel and removed the pointer concept of C++. This is regarded as one of the reasons why Java and Eiffel are better than C++. Figure 1 shows a simplified representation for objects of the class of persons in Java and C++.

<pre>class PERSON { private int Length; private String LastName; private final DATE BirthDate; private PERSON Spouse; // remaining instance variables }</pre>	<pre>class PERSON { private : int Length; char LastName[20]; const DATE BirthDate; PERSON* Spouse; // remaining data members }</pre>
---	--

Figure 1: Java and C++ representation for the class PERSON.

In Java, simple types such as int, float, byte, etc. adopt value semantics whereas class types adopt reference semantics. Like in C++, arrays and strings in Java adopt reference semantics. The major differences between Java and other OOPs in view of object representation are:

- As mentioned before, Java has removed the pointer concept of C++. This implies that the user of a class no longer decides on how to approach objects. Java has drawn the line between classes adopting value semantics and classes adopting reference semantics. In Eiffel on the other hand, the designer of a class chooses between value semantics (expanded classes) and reference semantics (ordinary classes). Expanded classes turn out to be very useful in dealing with objects such as dates, complex numbers, etc.
- Java is a pure OO language, and unlike C++, it does not support global variables: all data is declared within the context of a class. This is considered as a plus point in favour of Java as it is more object-oriented.

Our conclusion with respect to object representation is that Java did not actually bring new concepts to express the “has-a” relationships. Java does things in a different way, i.e., the decoration of classes is done in another way, but the ingredients are the same like in other languages. Java is better than C++ in that it removed the notion of pointer, but it is less expressive than Eiffel in that it lacks the notion of expanded classes. Note that in the original definition of Java there was no support for immutable (constant) instance variables as in C++. Fortunately this is corrected in the new release of Java where the final modifier can now be used with them as shown in figure 1.

2.1.2. Object Behavior

Now we turn to the issues related to the specification of operations applicable to instances of a class. These are called instance methods in Java, member functions in C++, and routines (part of the features) in Eiffel. In the OO paradigm, objects interact with each other by sending messages. While a C++ program could be written without creating objects and sending messages to them, Java expresses the message paradigm in a better way, thereby moving a step further towards pure OOPL. Some facilities offered by Java and C++ for specifying operations are given in figure 2.

<pre> class PERSON { public void ChangeLength (float newl) { Length = newl; } public boolean IsMarried () { return Spouse != null; } public void Link (PERSON partner){ Spouse = partner; } } </pre>	<pre> class PERSON { public : void ChangeLength (float newl)Length = newl;} bool IsMarried () const {return Spouse != null;} PERSON& Link (PERSON partner){ Spouse = partner; return *this; } // rest of member functions omitted } </pre>
--	--

Figure 2: Partial behavior for the class PERSON in Java and C++.

Figure 3 illustrates how methods in Java and member functions in C++ can be applied to objects of the class PERSON.

<pre> PERSON P1, P2; P1.ChangeLength(1.65); P1.Link(P2); if (!P1.IsMarried()) ... ; </pre>	<pre> PERSON P1, *P2; P1.ChangeLength(1.65); P2->Link(&P1); if (!P1.IsMarried()) ... </pre>
--	--

Figure 3: Manipulation of the state of objects in Java and C++.

We can see from figure 2 that C++ and Java offer the same facilities but differ in some points :

- The const-qualification at the end of a member function signature in C++ is removed in Java. This const-qualification specifies that the state of the object receiving the message can not be changed by the member function. This interesting feature is removed by Java.
- Unlike Eiffel features, Java methods do not specify assertions (expressed in preconditions and postconditions). We believe that the specification of operations applicable to objects of a class should be accompanied by the semantics of the operations.

Our conclusion with respect to object behavior is that Java did not bring new facilities for describing operations. Eiffel has a plus point over Java as it provides extra facilities for expressing the semantics of operations. In C++ one can prevent undesired changes to the actual arguments by a member function by qualifying the argument with const like in f (const PERSON& p). Again this feature is also added to the new release of Java. We believe that an adequate OOPL should offer decent concepts for restricting possible manipulations to information passed by clients.

2.1.3. Object Encapsulation

All OOPLs actually adopt the principle of information hiding. Important in this issue are the notions of :

- Class interface- specification of operations applicable to instances of a class.
- Class implementation- actual representation of objects plus algorithms for operations.

Classes introduce data and code and should protect them from other classes. Languages differ in their approach to this issue. In Smalltalk all instance variables are hidden from all clients, and all messages in a class definition are available to all potential clients. Java and C++ define different levels of access rights : public, private, protected, and friend. The friend modifier in C++ is a flexible mechanism which enhances efficiency by providing certain functions, or all member functions of a class, access to all the sections of a class. It is not supported by Java, but a similar behavior could be achieved by Java's protected statement in combination with packages. In Eiffel, attributes can only be queried (read-only) and the selective export construct caters for controlling access rights of features. The selective export mechanism of Eiffel is finer in the sense that access is granted just to features of interest rather than to everything like in the friend mechanism of C++. Packages in Java provide a sort of selective export- all non-private variables and methods of a class are visible to all other classes in the same package.

Separation of interface and implementation, as in C++, hampers maintenance of header files, especially when the system grows in size. One needs to make sure that they match the implementation and are kept with the correct version of the compiled class libraries. Therefore, the natural place for information about what is in a compiled class is in the same file. This approach is adopted by Java, Eiffel, and Smalltalk. These languages provide tools (like short in Eiffel) to extract the specification of the class. Both the include clause of C++ and the import statement of Java cater for class dependencies. In Java however classes are available by their fully qualified names and thus the use of the import statement is not a must. In languages like Eiffel and Smalltalk, the system searches for the classes. Thus there is no need to specify class dependencies, thereby making the issue more flexible than in Java and C++. However, when viewing Java in a network context, the explicit class dependencies becomes essential, since classes can be loaded across the network, and the scope of classes to load could be the entire network.

We can now conclude that Java is better than C++ with respect to the issue of separate interface and implementation, but in a sense less expressive than Eiffel with respect to the issue of access rights. Java's package is an interesting concept as it helps dividing classes that collaborate closely into units. This has many advantages as stated in [16].

2.1.4. Object Construction and Destruction

Constructors offer the ability to create new instances of a class, whereas destructors offer the ability to destroy existing objects of a class. Constructors in C++, Eiffel and Java are restricted to initialization of new objects. Only Smalltalk fully satisfies the semantics of constructors in the sense that instance creation methods are responsible for the creation and the initialization of new objects. Explicit destructors are supported only by C++ and their main usage is for clean up purposes. Constructors in these three languages have several aspects in common. For example, if no constructor is defined then default initialization rules apply (in C++ no default values). They all allow multiple constructors to be defined for the same class. The Eiffel constructor mechanism has some advantages over Java and C++, however:

- they have different names to suggest the purpose of each of them
- they can be exported as normal routines so that they can be called to reinitialize an object

The finalize method in Java provides similar functionality as destructors in C++ (releasing system resources before it gets collected). A finalize method is a useful concept which is, according to Java Language Specification 1.1, guaranteed to be invoked before the object is collected.

Java, Eiffel, and other languages such as Smalltalk and Beta appeal to garbage collection for reclaiming memory occupied by objects no longer referenced. Garbage collection is considered another reason why these languages are better than C++. In other words, programmers are relieved from the miseries related to memory management issues.

We conclude this section by saying that Java has advantage over C++ with respect to object destruction, since it uses garbage collection techniques to achieve it. As mentioned above however, Java is not the only language to provide such facility. The finalize method of Java is an attractive mechanism, but it is not clear that Java ensures its invocation always. With respect to object construction, Java is equivalent to C++ and Eiffel. However, the approach adopted by Smalltalk is purer in the sense that the construction of a new object confirms to the message paradigm. It is also more consistent in the sense that a constructor really creates a new object and belong to the metaclass. An interesting property of a java constructor is that it can call another constructor of the class or the superclass, using the this and super references with appropriate arguments to select the desired constructor.

2.2. Inheritance

Inheritance is the ability to create a new class from already existing class(es). Inheritance is used to express the "is-a" relationship between classes. A subclass (derived class in C++, heir class in Eiffel) inherits representation and behavior from the superclass (base class in C++, parent class in Eiffel). In fact all the languages deserving the description OOPs offer support for inheritance. However, they differ in their approaches regarding the relationship between the subclass and its superclass(es). They differ in, among other things, constructor invocation, access rights, access to superclass version of operations, name conflicts resolution, modification of inherited operations (redefinition), etc.

There are two kinds of inheritance, namely, single inheritance whereby the subclass inherits from only one superclass, and multiple inheritance (subject of section 3.2) whereby the subclass inherits from more than one superclass. There are several forms of inheritance [17]. In extension inheritance for example, the subclass introduces new data and/or operations not present in the superclass. At times

the subclass needs to keep the effect of inherited features, but wants to add extra actions applicable to its instances. All languages tend to support this type of inheritance in almost similar ways. However, they failed to truly meet the requirements of extension inheritance in its full meaning. The Beta language went a step further and allowed the implementation of an extended operation in the subclass to be combined with that of the superclass. By so doing, Beta becomes the first language to partially fulfil the requirements of extension inheritance [12]. This is actually the reason why we compare Java with Beta with respect to this matter. Note that not all operations or implementations are given in figures 4 and 5.

```

class POINT {
    private float x;
    private float y;
    public void move (float dx, float dy) { x += dx; y += dy; }
    public void clear () { x = 0.0; y = 0.0 }
}
class POINT3D extends POINT {
    private float z;
    public void move3D (float dx, float dy, float dz) { super.move (dx, dy); z += dz; }
    public void clear () { super.clear (x, y); z = 0.0; }
}

```

Figure 4: Extension inheritance in Java.

The inner construct in Beta is the mechanism that caters for the extension of behavior (combination of implementation). Whenever the procedure pattern move, for example, is invoked on an object of type POINT3D, the execution starts with move as defined in the super-pattern POINT. Then inner transfers control to the do-part (body) of the move in the sub-pattern to execute the extra actions. Note that, in Java, the use of super to call the version of a method in superclass is cleaner than the scope operator (::) in C++ which has many other purposes. Eiffel uses the rename clause to give access to the parent class's version.

Related issues to inheritance, among others, are: First, status of an operation in a superclass, that is an operation may be:

- abstract: must be effected by a subclass. This is supported by Java, Eiffel, and C++.
- final: should not be further redefined by subclass. Explicitly mentioned in Java using the keyword final. In Eiffel a routine that is qualified as frozen has the same effect. In C++ a non-virtual function implies a final one.
- extensible: may be extended (redefined) in subclass. This is supported by all languages.

```

POINT : (# x, y : @real;
    move : < (# dx, dy : @real enter (dx, dy)
        do x + dx->x; y + dy->y; inner
    #);
    clear : < (# 0.0->x; 0.0->y; inner #);
#);
POINT3D : POINT (# z : @real;
    move :: < (# dz : @real enter dz do z+dz->z; inner #);
    clear :: < (# do 0.0->z; inner #);
#);

```

Figure 5: Extension inheritance in Beta.

Second, the constructor relationship:

- Eiffel: subclasses should not invoke superclass constructors (default initialization applies). If they want to do that, then they should rename the constructors of the parent class.
- Java and C++: constructors should be explicitly called (default constructor otherwise).

Unlike C++, Eiffel, and Smalltalk, Java allows final classes that can not be further specialized. This is a limitation on the use of the class, but is useful in some situations. Beta supports this for virtual patterns.

Third, the access rights in a superclass can be changed in a subclass:

- Java: a subclass can change the access rights of a superclass's methods, but only if it provides more access, i.e., from protected to public.
- C++: functions and data members can be changed from public to private or protected.
- Eiffel: hidden features can be exported to be publicly available.

Fourth, redefinition of operation signature: Eiffel offers support for covariant signatures which means that the signatures of heir class do not have to match but to conform to those of the parent class. Java and C++ do not permit this, though it is a useful mechanism.

2.3. Polymorphism and Dynamic Binding

In fact polymorphism and dynamic binding are two concepts related to inheritance. The ability to associate with a variable objects of different types is called polymorphism. This is however restricted, in almost all languages, to types which are subtypes of the static type. Suppose we have the class hierarchy: PERSON<-EMPLOYEE<-MANAGER, then the next assignments in Java and C++ are legal: PERSON P; EMPLOYEE E; MANAGER M; P = M; E = M;

Assignments in the opposite direction (reverse polymorphism) are usually illegal: E = P; M = E; unless, as in Beta and in Java using a cast, at run-time P refers to an employee, and E refers to a manager respectively. The dynamic_cast function of C++ can be used to in achieve similar effect.

Dynamic binding is the ability to associate code with operation name at run-time. It is used to invoke the correct version of redefined operations. All the languages used in this article for the comparison include the notions of polymorphism and dynamic binding.

In fact, inheritance, polymorphism and dynamic binding are powerful tools supporting the OO paradigm. Like Eiffel and Smalltalk all non-final methods in Java are virtual, which gives Java an advantage over C++. By so doing, the decision as to whether or not a non-final method may be redefined in subclasses is left to the designers of subclasses. We believe that all these OOPLs deal with this issue in a similar way.

2.4. Abstract Classes

An abstract class is a class that does not have objects of its own. It contains actually general information common to several other (concrete) classes. For example figure 7 shows the Closed_Figure class which is abstract. Classes like POLYGON and ELLIPSE could be concrete ones specializing CLOSED_FIGURE.

<pre> abstract class CLOSED_FIGURE { abstract public void Perimeter (); abstract public void Surface (); } </pre>	<pre> class CLOSED_FIGURE { public void Perimeter () = 0; void Surface () = 0; } </pre>
---	---

Figure 7 : Abstract classes in Java and C++.

Abstract classes in Java are more readable and neat than in C++, though there is overhead in repeating the keyword `abstract` in the class and the abstract methods. Note that a fully implemented class in Java may nevertheless be declared `abstract`, what gives the designer some interesting design possibilities. Eiffel also supports the notion of abstract class (called deferred class). Conceptually, there is no difference in the support for abstract classes in C++, Eiffel, and Java.

2.5. Exception Handling

Although exception handling is not a required feature of OOPs, we include it here because it is becoming a characteristic of OOPs for which they provide several language constructs. Now all the three languages Java, C++, and Eiffel offer somehow similar built-in support for exception handling. The language constructs provided for exception handling are: the `try/catch/finally` in Java, the `try/catch` in C++, and the `rescue/retry` construct in Eiffel.

In principle, the `try/catch` statement of Java is similar to that of C++. Java added the `finally` block that contains clean-up code (close files, release resources, etc.). The `finally` block is guaranteed to be executed after the `try` clause which makes it a useful addendum to the `try/catch` construct of C++. The `rescue` clause of Eiffel can be compared to the `catch` statement of Java and C++ (both are referred to as an exception handler). The `retry` instruction within a `rescue` clause in an Eiffel routine allows the `do` clause of the routine to be executed again after some actions have been done. It is a useful feature that enables the implementation of the so-called alternative strategies. Both Java and Eiffel have a library class which includes information about an exception and can be inherited by any subclass.

We conclude that the three languages have almost similar support for exception handling. However, Java has a plus point over C++ by introducing the `finally` block, and a plus point over Eiffel by enabling the programmer to specify possible exceptions. It also distinguishes between compiler checked (general) exceptions and non-checked (run time) exceptions. Exception handling in Java is a more fine-grain mechanism than in C++.

3. Missing Desirable Features of OOPs

3.1 Genericity

C++ and Eiffel provide the ability to define parameterized classes. The notion of a generic class is just another means for reusability. Java does not include such a concept as the template of C++ or the generic classes of Eiffel. Figure 8 illustrates this concept as supported by C++ and Eiffel.

<pre> class QUEUE[ITEM] creation make feature make (n : INTEGER) is front : ITEM is add(i : ITEM) is end -- class QUEUE q : QUEUE[INTEGER]; </pre>	<pre> template<class ITEM> class QUEUE { public QUEUE(int n); ITEM front (); void add (ITEM i); }; QUEUE<int> q; </pre>
---	---

Figure 8: Eiffel and C++ generic classes.

Genericity can be simulated to some extent in Java using the Object class. In Beta virtual (class) patterns cater for genericity. Figure 9 illustrates these ideas in Java and Beta. In Beta, the type of the elements to be placed in a queue is declared as a virtual pattern extending the most general pattern Object. The problem with the queue of Java is that it can not hold primitive types (int, boolean, float, and such). However, Java offers wrapper classes (Integer, Boolean Float) that can be used in any context where an object reference is needed. The Beta solution is surely more elegant than that of Java. Thus with this respect Java has not provided good support as in other languages. In the context of typed languages however, the approach of C++ and Eiffel is to be preferred. It has the advantage of eliminating a number of possible errors at compile-time.

<pre> class QUEUE { public QUEUE(int n); public Object front (); public void add (Object i); } QUEUE q = QUEUE (10); PERSON p = new PERSON(); q.add(P); PERSON P1 = (PERSON) q.front; </pre>	<pre> QUEUE: (# Item : < Object; { type of elements } Init : < (# n : @integer enter n do ...#); Front : (# Result : @Item do exit Result #); Add : (# i : @Item enter i do#); #) PERSON_QUEUE : QUEUE (# Item ::< Person; ..#) pq : ^PERSON_QUEUE; </pre>
--	--

Figure 9: Generic classes in Java and Beta.

3.2. Multiple Inheritance

Unlike single inheritance, multiple inheritance (MI) is a controversial issue and there has been a considerable amount of discussion on its value. A thorough study of MI on its usefulness, difficulties and other related issues is given in [18]. Multiple inheritance, despite its complications, does add expressive power to and is a desirable property of the OO paradigm. It is also one of the most misused features of OO paradigm. There are several difficulties associated with MI one of which is the case when superclasses used for the derivation of the new class share common ancestor class. This of course leads to redundancy in representation.

OOPLs tend to avoid MI because of the extra difficulty in implementing it. C++ and Eiffel have offered support for MI to resolve redundancy and name clashes. It is not our objective in this article to explain the mechanisms offered by C++ and Eiffel to deal with this problem. Inheritance is mainly used for code reuse purposes, and MI can surely enhance reusability of code.

It is claimed that interfaces add MI to Java, whereby multiple interfaces may be inherited. In our opinion interfaces are similar to abstract classes that define a set of operations with no implementation. One can argue that the same effect of MI can be achieved with interfaces. The major problem with interfaces, however, is that they defeat the purpose of inheritance as a means for

reusability. This is because an interface provides no implementation, but must be implemented by each class that wants to use it. This is not to say that the notion of interface is completely useless and must be removed from the language. It is just to say that it is not as powerful as MI. We have not seen until now an example in literature that shows the usefulness of the notion of interface as a substitute for MI.

4. Additional Features of OOPLs

The rapid pace of technology evolution and thus the need for more complex software have led to an increasing interest in the incorporation of issues related to persistency, concurrency, and distribution into OOPLs. These concepts are not yet accepted as necessary requirements of OOPLs. However, we do hope that they will be accepted by the OO community, since these issues are becoming requirements of many types of applications. After all the OO technology is claimed to closely model the real world, and our world is persistent, concurrent, and distributed.

Java has done a pioneering work in these areas, especially the facilities offered to program the Internet. Java is actually unique in its support for distributed network-based applications. However, Java was not the first language to introduce such concepts. Beta ,for example, supports concurrency at the language and the programming environment levels. Several extensions to C++ were proposed to incorporate such issues in C++. Recently proposals are made to incorporate concurrency and distribution issues at language level in Eiffel [19]. A comparison of the language constructs and mechanisms offered by these languages is beyond the scope of this article. The solutions they provide are considerable improvements in these areas and form the building blocks for final solutions. It is possible that additional work is still required for a complete support, however.

5. Summary

The table given in figure 10 summarizes the main issues (common to all OOPLs) discussed in this article. Regarding the issues related to persistency, concurrency and distribution, Java has a rich programming environment and this gives it an advantage over the other OOPLs. Regarding the syntax, simplicity, readability, and portability of the language, we can say that Java is absolutely an improvement over C++.

	Java	C++	Eiffel
Notion of Object & Class	Yes	Yes	Yes
Object Representation			
<i>(Explicit) Pointers</i>	No	Yes	No
<i>Part (sub)Objects</i>	No	Yes	Yes
Object Behavior			
<i>Design by Contract</i>	No	No	Yes
<i>Constant Qualification</i>	Yes	Yes	No
Object Encapsulation			
<i>Selective Export</i>	Yes (package)	Yes (friend)	Yes
<i>Integral Interface/Impl.</i>	Yes	No	Yes
<i>Class Dependencies</i>	import statement	include statement	System Resp.
Object Const./dest.			
<i>Multiple Constructors</i>	Yes	Yes	Yes
<i>Garbage Collection</i>	Yes	No	Yes
Inheritance	Single	Multiple	Multiple
Generics	No	Yes	Yes
Exception Handling	Yes	Yes	Yes

Figure 10: Summary of the comparison.

6. Conclusions

On ground of the previous investigation, we can say that Java's hype and fame are not attributed to its object-oriented features. Java did not introduce new OO concepts or extended an existing one. In fact Java is a good synthesis of other major OOPs like C++, Eiffel, and Smalltalk. Java discarded interesting concepts from C++, namely, multiple inheritance and genericity. This is probably due to achieving one of its design goals, namely, simplicity. However, we think that this was at the expense of reducing the expressive power of Java regarding the OO paradigm. Also the notion of programming by contract is not supported by Java. The new features of Java in the areas of concurrency,

distribution, architecture neutral etc. are not yet accepted as essential characteristics of OOPs. If Java turns out to be important contribution to the OO software community, its support in these areas must be the reason for that.

References

- [1] Gosling and McGilton, The Java Language Environment, A White Paper, Sun Microsystems, 1995.
- [2] Arnold and Gosling, The Java Programming Language, Addison-Wesley, 1996.
- [3] Flanagan, Java in a Nutshell, O'Reilly & Associates, 1996.
- [4] Coad and Nicola, Object-Oriented Programming, Yourdon Press, 1993.
- [5] Winblad et al., Object-Oriented Software, Addison-Wesley, 1990.
- [6] Madsen *et al.*, Object-Oriented Programming in the Beta Language, Addison-Wesley, 1993.
- [8] Goldberg and Robson, Smalltalk-80, The Language and its Implementation, Addison-Wesley, 1983.
- [9] Thomas and Weedon, Object-Oriented Programming in Eiffel, Addison-Wesley, 1995.
- [10] Meyer, Eiffel : The Language, Prentice-Hall, 1992.
- [11] Stroustrup, The C++ Programming Language, Second Edition, Addison-Wesley, 1994.
- [12] Al-Ahmad, Steegmans, Specialization of Behavior : Comparison, Critique, and A New Approach, Accepted for publication in JOOP, 1996.
- [13] Joyner, C++ ?? A Critique of C++, 3rd Edition, <http://www.csd.uu.se/~alex/study/cppv3.ps.z> , 1996
- [14] Kristensen and Østerbye, A Conceptual Perspective on the Comparison of Object-Oriented Programming Languages, ACM SIGPLAN Notices, Vol. 31(2), 1996.
- [15] Berard, Essays on Object-Oriented Software Engineering, Prentice-Hall, Vol. 1, 1993.
- [16] Rumbaugh, Packaging a System: showing architecture dependencies, JOOP, Vol.8(7), 1996.
- [17] Budd, An Introduction to Object-Oriented Programming, Addison Wesley, 1997.
- [18] Singh, Single Versus Multiple Inheritance in Object-Oriented Programming, OOPS Messenger, Vol. 5(1), 1994.
- [19] Meyer, Concurrency, distribution, client-server and the Internet: <http://www.eiffel.com/doc/manuals/CONCURRENCY.html>, 1996.