Journal of Computer Sciences 1(4): 450-453, 2005
ISSN 1549-3636
© Science Publications, 2005

A Note towards Reshaping Java's Features

N.S. Kutti, Z.A. Al-Khanjari, H. A. Ramadhan and J. Fiaidhi
Department of Computer Science, Sultan Qaboos University,
P.O. Box 36, Al-Khodh 123, Sultanate of Oman

Abstract: Programming Languages continually go through the refinement process due to several findings such as lack of robustness, lack of flexibility, lack of simplicity and lack of standard. This may require pruning undesirable features, sometimes softening of hard nature of some features and adding new features to improve the scope of the languages. Recently Java has emerged as a refined language in the line of C and C++ with the aim of providing simplicity and robustness[1]. Because of these features Java is getting more attention than its predecessors. The language is, however, overshadowed with some inconsistencies in the syntax and semantic aspects of data declarations. The study also identifies several other redundant features that could be safely removed from the language. The aim of this study is to underline these findings that make java programs somewhat obscured. The discussion carried in this study may be an useful hint for the Java reviewers as well as any new language developers in validating their specifications. Java has definitely a long future and its current review will extend its scope even to support hard real-time applications[2].

Key words: variables, lvalue parameter, rvalue parameter, pointers, objects, data memory, heap storage

INTRODUCTION

Java has evolved as a refined object-oriented language from its predecessor, C++. Because of its simplicity and versatile nature it is gaining its popularity as a general-purpose language within the computer Science community. Java developers must be praised for retaining the most of the C/C++ syntax and removing several unreliable and unnecessary features of C/C++ at the same time[1]. While more attention was paid towards simplifying hard nature of C++, it seems that some inconsistencies have somehow escaped from the attention of the Java developers. The inconsistencies are noticed in several forms. This study discusses the lack of standard in reference declaration, mixing C++ convention with object declarations and inheriting C++ features without any concrete reasoning. This study may be a useful hint for the programming community in general.

Inconsistent notation with object declaration: A language usually provides one or more means of accessing data from memory. The most common way of accessing a stored data is by using the variable concept. A variable by definition is a direct reference to a stored data in memory. It comes with lvalue and rvalue parameters[3]. The lvalue parameter acts as a frame and the rvalue parameter acts as a picture in the frame[4]. A variable is nothing but framed picture (Fig. 1). The variable name is a reference or a symbolic address created to locate the frame. When we use the reference in the left hand side of an assignment operator it represents the address of the frame and in the right side it represents the picture (or value) that it holds. Alternate mode of accessing data is through indirect means. In this mode, data is accessed via an address held in a pointer. There are basically two types of pointers: *primitive* and *abstract* pointers. A primitive pointer can hold the physical address of a variable in memory. Both C and C++ support primitive pointers particularly to cater for system programming. An abstract pointer on the other hand holds a frame address or simply a reference of a variable. Hence, an abstract pointer can be referred to as either frame pointer or reference pointer. Pascal, Modula-2 and Ada use this mode of addressing for managing dynamic data structures. Java also uses the frame pointer concept exclusively to manage all objects. Using a convenient notation:

```
INTEGER :: K                   // DECLARE AN INTEGER VARIABLE
INTEGER (MEMORY):: *MP         // DECLARE A MEMORY POINTER
INTEGER (FRAME)::    *FP       // DECLARE A FRAME POINTER
*MP = &K                       // ASSIGN MEMORY ADDRESS OF K
*FP = K                        // ASSIGN FRAME ADDRESS OF K
```

Corresponding Author: J. Fiaidhi, Department of Computer Science, Lakehead University, 955 Oliver Road, Thunder Bay, Ontario P7B 5E1, Canada E-mail: jinan.fiaidhi@lakeheadu.ca

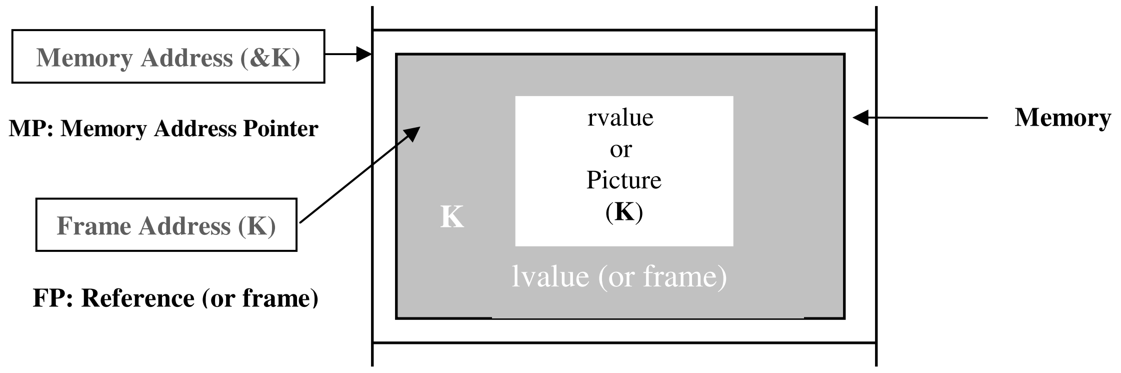

Fig. 1: Concept of Variable

Table 1: Declaration syntax for data variables and pointers

Language	Normal Variable	Address(or Primitive) Pointer	Reference (or Abstract) Pointer
C	int k;	int k;	None
		int *pk;	
		pk = &k;	
		*pk = 100;	
Pascal	k: integer;	None	var
			ip1, ip2: ^integer; (* declare null FRAME pointers *)
			. . .
			NEW(ip1); (* create a variable and assign its frame to ip1*)
			NEW(ip2); (* create a variable and assign its frame to ip2*)
			ip1^ := 99; (* assign 99 to ip1's reference variable*)
			ip2^ := 1; (* assign to ip2's reference variable*)
			ip1 := ip2; (*both ip1 & ip2 point to same frame, ip2^ *)
Modula-2	K:INTEGER;	None	VAR (* declare IP as a null integer pointer *)
			IP : POINTER TO INTEGER; (* create a frame pointer *)
			. . . .
			ALLOCATE(IP,TSIZE(INTEGER)); (* create a variable and assign its frame to IP *)
			IP^ := 27; (* assign a value to IP's variable *)
Ada	K:INTEGER;	None	type Node_ptr is access Node; -- declare a Node_ptr type
			type Node is -- declare Node type
			record
			value: integer;
			next: Node_ptr
			end record;
			np: Node_ptr; --
			np := new Node; construct a node
			np.next := null
C++	int K;	int *p;	int j, k; // y is simple variable
		X_Class:*ptr_to_obj;	int &jj = j; // declare an alias to frame j
			j = jj; // same as j =j; or jj=jj; j and jj refer to same frame
			X_Class x; //x is an object
			X_Class **&**x_ref = x; //a constant reference to x

It is conceivable that when a language supports more than one type of access (i.e. direct as well as indirect), there must be a clear distinction between the declarations of these two types. This distinction is usually achieved by maintaining separate syntax for each type of declaration. In fact, this has been normal tradition followed by almost all languages. As an example, Table 1 shows the notations adopted in some of the popular languages.

From Table 1[3,5-9] we notice that Pascal, Modula-2 and Ada implemented reference pointers without any ambiguity. They use a special syntax to declare a reference variable. Similarly, C clearly distinguishes declaration of a memory pointer from the declaration of a normal variable declaration. Taking the case of C++ declarations of all four types (i.e. normal variables, objects, memory pointers and reference-aliases) show clear distinctions and avoid any misinterpretation. In Java, a deviation is detected in the case of declaring reference pointers. The declaration syntax of a reference pointer looks like declaring a normal variable. The consistency maintained by C and C++ is somehow lost in Java. See the following declarations in Java:

```
int   k;    // k is an instance of integer type, i.e. a variable
class_X  x;   // x is not an instance of class_X, but a reference pointer to class_X . . .
x =  new class_X;  // create an instance of class_X and
     assign its frame address to x
```

Finally, distinguishing the types of entities created in these two declarations and interpreting their meanings are entirely left to the intuition of the programmers. Even experienced programmers may find it difficult to interpret an ambiguous situation created by the language implementation. According to implementation the new operator returns an address of created object and this returned address can be assigned to a reference declared as per Java's notation. According to the original definition the term "reference" means the actual lvalue (or an alias to this lvalue) of a variable and it can only be assigned with an rvalue of the same type. Therefore, assigning an address to a reference is an ambiguous expression.

Informal notation in passing parameters: Java encourages an informal protocol while passing parameters to methods. Unlike in C++, the Java compiler has to interpret which formal parameter is passed as a value and which one as a reference. Java assumes that a parameter of primitive type is always passed as a value and an object as a reference. While Java encourages strong typing on one hand, it fails to support a strict discipline in the explicit declaration of formal parameters in the method definitions. This is another inconsistency in Java. One reason for this inconsistent convention is due to the omission of declaration of constant reference variable (or declaring an alias to reference) of primitive types. Particularly, after omitting both the memory-based pointers and aliases to references in the inherited Java there is no way that a method can use parameter passing by reference for primitive variables. Java's assumption that all simple variables are going to be global within an object may not be strictly true. We should not forget the presence of local variables in a method.

Inconsistency with array declaration: Unlike in C and C++ arrays in Java are created as objects. That is, an object creation takes two steps: a reference pointer to array class is first declared and then the reference pointer is made to point to frame of the object created in the heap memory. These steps are expressed in different ways as shown below:

```
(a)  int[]  arr; // declare a reference pointer
     arr = new int[10]; //  create an array object and
     assign its frame address to arr
(b)  int[]   arr = new int[10]; // declare a reference
     pointer and assign frame address
     // of the created array object
(c)  int   arr[] = new int[10]; // same as (b) style, but
     with C/C++ style
```

In both (a) and (b) declarations, int[] acts as a key word for integer-array class. First arr is declared as a reference pointer of int[] type, secondly the new operator returns the address of created array object in the heap memory and finally the lvalue (or frame address) of the created object is assigned to the reference pointer. But the syntax adopted in (c) shows a bad influence of C/C++ on Java. Java programmers with C/C++ background may tend to interpret the use of square brackets with the reference pointer as an array of reference pointers rather than a reference to the array type. On one hand Java does not recognize a construct such as

```
int  arr[10] = new int[];
```

and on the other hand allows similar syntax shown in (c). Java should, in fact, stick to the syntax adopted in (a) and (b). That is, the notation "int[]" will suffice to indicate the integer array class.

Another inconsistency is also noticed in the array declarations. Consider the following Java declaration of an array object with initialization:

```
int[]  arr =  {0,1,2,3,4,5,6,7,8,9};
```

Again this implicit notation happens to be in Java because it is simply inherited from C/C++. But it can be interpreted as if an initialized object is created in data memory. Similar ambiguous syntax is adopted for declaration of character arrays and String objects with initialization. In fact, Java uses the "new" operator explicitly to indicate that the class object is created in the heap storage. This shows a further deficiency in the clarity of declaration syntax. As a result a C/C++ programmer may tend to interpret that Java can create objects in heap as well as in data memory. Java has failed to maintain the consistency in pronouncing the concept of "Java Object" to the programming community.

Array bounds: Java simply inherits the C/C++ convention in specifying the array boundaries. Declarations require only the sizes of arrays as their dimensions. The lower boundary of a dimension is always assumed to be 0. This feature became significant in C/C++ due to the requirement of several machine level programming features to support system programming. But Java has unnecessarily adopted C-convention in this respect and created a confusion among application programmers. Java could have adopted either a simple abstract notation with the lower boundary, as 1 or more structured notation as in Ada to avoid programmers making any wrong assumption on the boundaries. Java with similar C-conventions may require a previous knowledge of C/C++. It creates somewhat a soft constraint on the learning of Java!

Redundant data types: Java has unnecessarily retained the *short* data type. This integer category was introduced in C due to non-standard definition of *int* category. Portability issue with integers worked very well with *short* and at the same time *short* was tagged as a memory saving integer type. With the current VLSI technology almost all microprocessors have settled with 32-bit word length and the cost of main memory has no barrier any more to have almost unlimited memory with the current processors. Because of this factor concept of using *short* has become obsolete.

The *long* data type can be merged with the 64-bit *int* category to satisfy all applications without any loss of generality. This may not pose any considerable problem in embedded and other low-end applications. Similarly, *double* gives a redundant feature over float. The *float* type can also be made to provide a wider role without any loss of generality. Such reductions will further simplify the language and improves the portability of written programs.

Useful data types: The unsigned data type in C/C++ is used for addressing memory particularly in system programming. Java developers in the process of simplifying the language have omitted not only memory pointers but also unsigned integer. The unsigned discrete data type can be useful in driving embedded real-time applications. Several abstract data structures such as stack and hash table can use unsigned integers to emulate memory addresses such as stack pointer and hashed address. Another minor note is on the use of the *final* key word in defining data constants. The *const* key word has been traditionally used in many languages (e.g. Pascal, Modula-2, C and C++) to define data constants. It is a mystery how Java ignored to inherit this feature from C/C++. The keyword *final* is rather more appropriate as an operator for freezing variables or methods at particular state.

CONCLUSIONS

C was, in fact, an unbeatable system programming language until late eighties. Then C ++ with its object-oriented feature continued the C's role. Both languages enjoyed their importance in system programming because of the powerful pointer addressing. C++ became too cumbersome to handle because of too many additions over its predecessor, C. After C and C++, Java has evolved as a refined, simple and robust language. Subsequently Java has been tested for its level of robustness and reliability[10]. Since the study focuses on the lack of consistency it cannot spare any comment on the robustness aspect of java.

According to Java's declarative syntax with reference and the new operator creates a type conflict. In other words, the concepts of reference pointer and new operator need proper recognition. If Java had inherited the dynamic object declaration based on the reference pointers as well as object declaration in the data memory of programs, it would be as versatile as C++. The garbage collection involved with dynamic memory objects may not be suitable for some applications. For instance, use of Java coming with its garbage collection facility in real-time applications may have some influence on the unpredictability factor in real-time scheduling.

The redundant notations like *short* and *long* can be removed or retained as aliases for 64-bit *int* type. Similarly, *float* and *double* can be synonyms for a standard real type. This would reduce a constraint on type selection and in turn normal application programmers will feel less burden on their programming tasks. A pruning is needed in array declarations. A conservative syntax specification should be maintained for array declarations with initialization. At the same time, Java can also extend the array declaration syntax to include the specification of lower and upper boundaries without affecting the existing convention that can be treated as a special case. These would not only make the language more sound but also robust.

Java has taken a right step in upgrading the character size to 16-bit for the bigger Unicode character set. The *const* qualifier can be introduced for declaring constants without any side effect on the language.

Now Java has lost the pointer facility, the most useful mechanism for system programming. This has created a situation where C and C++ can still be considered as unbeatable language tools for system programming applications. In the evolution of Java, pruning C++ has created C++--- rather than transformed into a graceful C+.

REFERENCES

1. Singhal, S. and Binh Nguyen, 1998. Java factor: Introduction. Communications of the ACM, 41: 38-42.
2. Bollella, G. *et al*., 2000. The real-time specification for java. Addison-Wesley, Boston.
3. Kernighan, B. and D. Ritchie, 1991. The C Programming Language. 2nd Edn., Prentice-Hall, NJ.
4. Kutti, N.S., 2002. C and Unix programming: A comprehensive guide with ANSI and POSIX standards. Light Speed Books, Mt. Pleasant, SC.
5. Dale, N. and Chipp Weems, 1987. Pascal: D.C. Heath and Company, Lexington.
6. Wirth, N., 1982. Programming in modula-2, springer-verlag, 2nd Corrected Edn.
7. Barnes, J.G.P., 1998. Programming in Ada-95, 2nd Edn. Addison-Wesley.
8. Stroustrup, B., 2000. The C++ programming language. Addison-Wesley, NJ.
9. Arnold, K. and James Gosling, 1998. The java programming language, 2nd Edn. Addison-Wesley, Reading, MA.
10. Hunt, J. and F. Long, 1998. Java's reliability: An analysis of software defects in java. IEEE Proc. Software, 145: 41-50.