Jr: a Language Designed for Beginners

(Less Is More)

John Motil
Computer Science Department
California State University
Northridge, CA 91330
jmotil @ csun.edu

David Epstein
Computer Science Department
California Institute of Technology
Pasadena, CA 91125
depstein@cs.caltech.edu

Abstract

This paper describes a way of teaching programming to beginning students, by first introducing them to basic principles with a small language having a simple syntax and student-friendly environment which runs in a browser. Shortly after the basics, students can convert quickly and easily from Jr to Java. Less at first results in more later.

Introduction

Jr is a programming language designed for beginners. It has minimal punctuation; no semicolons or curly braces to misplace. It does not have inheritance; inheritance is not typically introduced in the first half of a beginning programming course. It has only one-dimensional arrays; higher dimensions can be created as classes. It aims for there to be only one way to do anything: one constructor, one integer type, one real type, one kind of comment, one repeat form, one choice form, etc. By avoiding the "many" ways to do similar things, Jr avoids considerable confusion, and saves an amazing amount of time. This time can be spent on principles rather than details.

Partway through the semester (before the half-way point) Jr is transferred into Java, rather quickly and simply, either manually or automatically (in an environment). During this transition the previous principles can be reviewed in Java, with more detail (showing two kinds of reals, three kinds of comments, four kinds of integers, and five ways to increment, with their associated problems). Then more material can be covered (uncovered) using Java.

Jr also includes "pure" functions (without side-effects), assertions (preconditions, postconditions and class invariants), and exception handling; in essence, programming by contract. Since the Jr language and environment is built for beginners, the time saved compared to using professional languages and environments can be used to consider these disciplined programming concepts.

After introducing Jr, this paper also shows another way to describe syntax using a visual nesting of boxes of various kinds.

In summary, by using Jr to initially avoid distracting details, more material was covered, it was covered faster, and ultimately better (because of the second pass over some material). This approach put the principles and the practice into proper perspective.

This Jr language was designed in the spirit of Blaise Pascal, who once wrote that he would have written a shorter letter if he had more time.

1. WHY: The Problem(s); More is Less

Most programming languages that are being used in beginning classes are large and complex. These languages are full featured, with industrial strength development environments. Even the formerly little language of Pascal has been abandoned by many suppliers, or morphed into a larger language (Delphi). Similarly, the modest Modula has grown into the bigger "Black Box" (formerly Oberon). Even Basic is big (Visual Basic). This leaves very few small languages for beginners[1].

Some students learn readily to use the huge environments, editors, debuggers, workarounds; but many of the skills are not "portable" and take time away from principles. They also may give wrong impressions; for example, no matter what principles the instructors may try to teach, the computer (compiler) seems to tell the students "It's the syntax, stupid!".

Previous experiences of teaching large languages suggested that many beginners have problems with the sheer size and complexity of such large languages. Even the better students can become frustrated and change their major to mathematics, business, or something else. So we were careful at CSUN when introducing Java for the first time. The results, however, were lower than expected. We had not covered as much material as anticipated, and the students seemed to understand that smaller amount less. This consensus was confirmed by a pretest given at the beginning of the following course.

Students felt that they learned a lot, but much of it involved details which were not fundamental. This second course used another platform, and student's "knowledge" did not transfer readily. Some students

forgot the fundamentals but remembered the details associated with the previous platform and environment! Unfortunately it is often easier for them to remember details than to digest concepts.

2. HOW: The Evolution of a Solution

In the dozens of years of teaching beginners at CSUN we used various languages, from Fortran to Pascal, to Modula, to C++, back to Modula and are now trying Java. We often introduce basic concepts visually, using some diagrams (class diagrams, flow blocks, datagrams, sometimes even flow charts) and use a simple pseudo-code for "linear" communication before coding into a "real" language. This is quite effective as the pseudo-code prevents premature coding with its emphasis on syntactic details rather than basic principles. However, students often resist the pseudo-code step, jump directly into coding, and get bogged down in details.

The Solution: Jr

The solution seemed simple; program the pseudo-code! But it took two semesters to work out the details of essentially another language. Design of a language, even a small one, was not a simple process. Design by two people probably doubles the time but results in a nice creation of combined ideas! Then, what seemed simple to us was not always simple for students, so the language went through many refinements.

During the first semester, Jr was used mainly as a pseudo-code, strictly on paper. The constructs were such that each line of Jr converted to one line of Java code. But students did this conversion line by line manually, an error prone process. Jr was at that time essentially a subset of Java, but Java was still "lurking" in the background, providing weird error messages and tempting students to venture outside of the subset, often wasting valuable learning time.

By the second semester of this experiment the language was rather refined. Also, a programming environment was created to convert the Jr code to Java code, automatically. The environment has two panels side by side; Jr code is entered into the left panel, and it is converted automatically (and correctly) into Java in the right panel. Any errors in the Jr programare displayed below, in a bottom third panel; only one error is shown at a time. When the conversion is completed (we say that the "Java is brewed")students can compile and run this Java code on whatever Java system they wish; we used various versions of the Sun JDK. In later versions the Jr code is rundirectly from the web.

A typical conversion showing the one-to-one side-by-side similarily follows:

```
If C then          if (C) {
   A                   A;
Else               } else {
   B                   B;
   D                   D;
EndIf              } // EndIf
```

Notice that in the above case (involving a single action) the curly brackets in the Java code are not necessary, but such extra braces are not removed in order to avoid the dreaded "dangling-else" problem. The dangling refers to the "else" as in the following Java:

```
if (a) if (b) x; else y;
```

where it is unclear which of the two ifs corresponds to the single else. Beginners don't even notice that there is a problem here. This problem cannot occur in Jr, but it plagues C, C++, Java, and even Pascal.

Use of the Jr environment resulted in less emphasis on Java (every line of Jr was converted into one line of Java, correctly), so students did not need to be concerned about the syntax of Java. Even though they didn't write the Java (yet), students saw the Java code, they understood it, usually, but they did not need to get into the details of Java directly at an early stage. This actually saved considerable time. Less is more!

Beginners often "imprint" to their first language or paradigm, and if it is too strongly engrained in them they may find difficulties in changing to another. To prevent such a "single mindset" we introduce two paradigms. First we briefly teach the procedural (imperative, top-down, control-flow, action-oriented) with the simple choice form and loop form. We are careful not to spend too much time on algorithms with complex nests of choices and especially loops. Then we quickly cover some data-flow concepts to introduce methods (functions and routines) and encapsulation, before introducing object-oriented programming. Admittedly, this multi-view may be somewhat confusing to beginners, but it prevents perverse preferences in the long run. This constitutes real teaching versus only training.

3. WHAT: The Language; Jr

The language Jr is small and simple. It has very few punctuation symbols. It has no colons, semicolons, curly brackets, or other such symbols which are extremely common in most other languages (and often have have many meanings in a language). Since it has no semicolons only one instruction is permitted on any line; this is not a big problem for beginners. If they don't have semicolons or curly braces they cannot misplace them! Less is more!

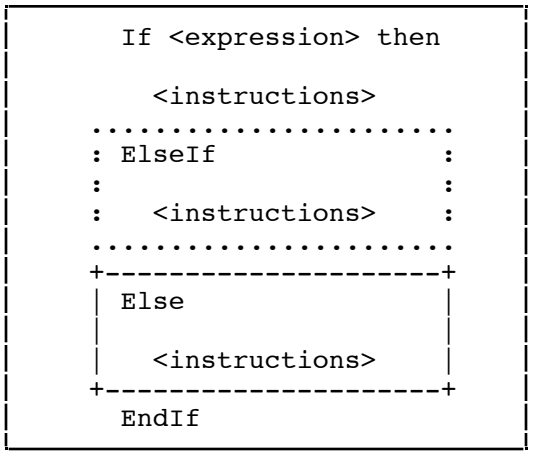
Uniqueness is a special feature of Jr, in the sense that there is only "one" of most things; one integer type (an int, rather than a short, a long, a byte, etc.). It has only one real type (rather than a float and a double). It has one string type (not both a string and a character type);

characters are simply short strings. There is only one Repeat form and one Choice form (an If, but no case form). It allows only one constructor, not many as in most modern object oriented languages. There is only one kind of comment, not 3 as a Java. There is only one way to increment. Later, after converting to Java, students can learn many (5 or more) ways to increment, and the associated problems. For beginners, one way is sufficient; more ways, and reasons to choose among the many different ways, can come later.

Jr has no BEGIN-END (like Pascal) and no curly {} braces (like C or C++) because each form, such as the If has a corresponding EndIf, each Class has an EndClass, each Function has an EndFunction, etc. This bracketing or "sandwiching" has a consistency and simplicity which is appealing to students. This kind of construct also prevents problems, such as the "dangling-else" problem (previously mentioned).

3.1 New Syntax Notation; nested boxes

Syntax of a language can be expressed in many ways (BNF, EBNF, syntax diagrams, etc), but yet another way was discovered here. It consists of a kind of template, skeleton or outline of nested boxes. Boxes with dashed lines surround those parts that are optional, i.e. can appear zero or one time. Dotted boxes surround those parts which can be repeated any number of times (including zero). In the following syntax definition of the Choice form, the "ElseIf Part" can occur often, whereas the "Else Part" can occur once at most. The number of occurrences is suggested by the number of symbols (few dashes, many dots); that is very easy for beginners to remember



Repeat is the only loop form; it allows just one exit from the loop which can occur at the beginning, at the end, or in the middle as long as it is not inside a Choice form. Using this one Repeat form, it is easy to create While, Until and For loops. The syntax blocks defining the Repeat form follow.

```
Repeat
   <instructions, if any>
ExitOn <boolean expression>
   <instructions, if any>
EndRepeat
```

It has been shown (by Soloway [4] and others [3]) that such a repeat construct which has a "closer cognitive fit" with individuals preferred cognitive strategy is easier to use effectively and is less prone to error.

An important goal of Jr is to be a beginner's language, and to move on to Java quickly. For this reason various limits were imposed on the language. Data structures are limited in Jr. it has only single-dimensional arrays; there are no arrays of two or more dimension. However, the class structure of Jr can be used to easily create multi-dimensional arrays. Growing larger programs is done using composition and the has-a relation; inheritance, the is-a relation is left for Java. Inheritance seems too complex a concept to introduce at too early a stage.

Jr includes many intrinsic or built-in functions (involving strings, trigonometry, etc) which in Java are more awkwardly called from various Java classes (involving the name of the classes, such as Math.sin(X)). However, this language does also allow the importing of classes from Java.

Input and output in Junior are most convenient for beginners. Most introductory Java texts create their own input libraries (with a method for each type, such as ReadInt, ReadDouble, ReadString, WriteInt, WriteFloat, etc), whereas Jr has a single "Input x" instruction for whatever type x may be.

Very few new words are introduced in this language. The name "box" is used instead of variable, to avoid confusion with the mathematical concept of that name. This keeps the newer, more dynamic concept from interfering with the older "static" concept. The word "slot" is used instead of parameter or argument, and it is represented graphically as a slot.

The language is both textual and graphical; it appeals both to the left and right brain. The two-dimensional nature of the code is emphasized by indentation. Indentation in general is not required, but it is required that the If and EndIf be indented the same amount; they must begin in the same "column" (and similarly for the Repeat and EndRepeat, the Class and EndClass, etc).

Documentation in the form of a single "Does" line is required in every class and every method (function or routine). A name of the author is also required at the beginning of every class. Other documentation can be in the short form of comments following a double-dash to the end of the line.

Beyond Java: Unmodifiable Public Data, Pure Functions, Assertions, Exceptions, and Programming by Contract

Java sometimes gives beginners too much freedom, leading students to develop poor programming habits. Jr imposes some limits on what can be accepted. For example, a public "box" cannot be modified outside the class in which it is declared. That is, the following is not allowed:

```
a.b = 3; // this is allowed in Java
```

The same code in Jr

```
Set a.b = 3 -- this is not allowed in Jr
```

reports an error.

All functions in Jr are defined to be "pure", which means that they have no side-effects. Functions in Jr cannot both return a value and also modify an object. For example, in the proverbial stack example, the pop method both returns the top value and modifies the stack position. Instead, there are two separate methods, a "top" function to return the top, and a "pull" routine (procedure) to modify the stack.

Programming by contract [2] is possible in Jr, but not required. It involves assertions in the form of preconditions, postconditions and a class invariant. Preconditions and postconditions are done using PreCheck and PostCheck instructions. The class invariant is done using Check instructions inside Invariant and EndInvariant instructions. A Check instruction is of the form:

```
Check <condition> bounce <string>
```

Similarly, PreCheck and PostCheck instructions are of the form:

```
PreCheck <condition> bounce <string>
PostCheck <condition> bounce <string>
```

Since Jr functions are free of side-effects, there is no reason to call the class invariant when returning. A public Jr routine, however, calls the class invariant (if one is supplied), before returning. Since private Jr routines are not allowed to call public Jr routines, there is no messy handling of calls to the class invariant (for example, a public routine can call a private routine without needing to reinstate the class invariant).

For example, in an Account class the routine debit which withdraws a given amount could have the two pre-conditions:

```
PreCheck (amount >= 0.00)
   bounce "Negative debit!"
PreCheck (amount < balance)
   bounce "Insufficient amount!"
```

A simple exception handling facility is also provided, mostly to teach the concept of exceptions. There are two instructions in Jr use to introduce exception handling, TryCall and TrySet. There is a Jr file I/O library which offers, for example, a readLine() method on a JJInFile object. One is not allowed to call this method with the Jr Call instruction. Instead, the TryCall instructions must be used. If the readLine fails, the TryCall instruction is used to either set a box or call a method. For example:

```
TryCall aReadFile.readLine()
    onFail badRead = true
```

The TrySet instruction is used when attempting to convert a string to an integer. Upon failure, it also can set a box or call a method. For example:

```
TrySet anInt = StrToInt(s)
     onFail reportBadIntString with (s)
```

Notice that calling a method uses the word "with" if there are any "slots" passed to the method.

4 WHERE:

The environment for Jr, with teaching materials, is available on the web at

www.publicstaticvoidmain.com

Summary

Using a simple language initially allows beginners to avoid distracting details, and results in more material being covered, at a faster pace, and ultimately better (because fo the second pass over some material). Less at first is more later.

References

[1]Kolling, M and Rosenberg, J. Blue - A Language for Teaching Object-Oriented Programming. SIGCSE Bulletin, 28, March 1996, 190-194.

[2] Myer, B. Object-Oriented Software Construction, Prentice Hall (1997).

[3] Roberts, E. Loop Exits and Structured Programming: Reopening the Debate. SIGCSE Bulletin, 27, March 1995, 269-272.

[4] Soloway E., Bonar, J. and Ehrlich, D. Congnitive Strategies and Looping Constructs: an Empirical Study. in Soloway, E. editor, Studying the Novice Programmer. Lawrence Erlbaum Associates, publishers, 191-207.

The following class, describing a bank account, shows many aspects of the language Jr. Some of the names have been shortened so that they fit into the narrow column.

```
Import JJIO
Class Account

-- Name Ann Onymous
-- Does provide a simple savings bank account
-- Shows much of the programming language Jr
-- Shows PbC, Programming by Contract

-- Data attributes, fields

   Box balance  ofType real is private
   Box identity ofClass Str is public

   Invariant
      Check (balance >= 0.00)
         bounce "Negative balance!"
      Check (identity != null) bounce "Null id!"
   EndInvariant

   Constructor Account (b, i) is public
      Slot b ofType real -- account balance
      Slot i ofClass Str -- identification
   -- Does initialize or open the account
      PreCheck (b >= 0.00)
         bounce "Negative initial balance!"
      PreCheck (i != null)
         bounce "Null initial identity!"
      Set balance = b
      Set identity = i
   EndConstructor Account

   Routine setBal (amount) is public
      Slot amount ofType real
   -- Does set or reset the amount of balance
      PreCheck (amount >= 0.00)
         bounce "Setting negative balance!"
      Set balance = amount
   EndRoutine setBal

   Function getBal (none) ofType real is public
      Box result ofType real
   -- Does return balance in account
      Set result = balance
   EndFunction getBal

   Routine credit (amount) is public
      Slot amount ofType real
   -- Does deposit an amount into account
      PreCheck (amount >= 0.00)
         bounce "Negative deposit!"
      Set balance = balance + amount
   EndRoutine credit

   Routine debit (amount) is public
      Slot amount ofType real
   -- Does withdraw an amount from account
      PreCheck (amount >= 0.00)
         bounce "Negative debit!"
      PreCheck (amount <= balance)
         bounce "Insufficient balance!"
      Set balance = balance - amount
   EndRoutine debit

   Function canCover (amount) ofType bool is public
      Slot amount ofType real
      Box result ofType bool
   -- Does check if balance covers amount
      Set result = (balance >= amount)
   EndFunction canCover

   Routine compound (percent, duration) is public
      Slot percent ofType real -- percent rate
      Slot duration ofType int -- time duration
      Box rate ofType real -- interest rate
   -- Does compound balance at a rate for a time
      PreCheck (duration >= 1)
         bounce "Negative duration!"
      Set rate = percent / 100.0
      Repeat
      ExitOn (duration == 0)
         Set balance = balance + balance * rate
         Dec duration by 1
      EndRepeat
   EndRoutine compound

   Routine getFrom (account, amount) is public
      Slot account ofClass Account
      Slot amount ofType real
   -- Does transfer from one account to this
      PreCheck (account.balance >= amount)
         bounce "Insufficient balance!"
      Call account.debit with (amount)
      Call credit with (amount)
   EndRoutine getFrom

   Routine show (none) is public
   -- Does display the account
      Output "Account " + identity + " has balance "
      Outputln balance
   EndRoutine show

   Routine test (none) is private
      Box his ofClass Account
      Box her ofClass Account
      Box amount ofType real
   -- Does test Account class
   Start
      New his ofClass Account with (100.0, "Dad")
      New her ofClass Account with (  0.0, "Mom")
      Call his.credit with (200.00)
      Call his.compound with (10.0, 8)
      Call his.show
      Output "Enter her deposit "
      Input amount
      Outputln amount -- echo
      Call her.setBal with (amount)
      If his.canCover (600.00) then
         Call her.getFrom with (his, 600.00)
      Else
         Outputln "No transfer possible"
      EndIf
      Call her.debit with (300.00)
      Output "Her balance is "
      Outputln her.getBal()
   EndRoutine test

EndClass Account
```