Reference for the Programming Language JJ

JJ, or Junior Java, is a language designed for beginners; it is not a "toy" language, but is a gentler version of a "senior" language. It is meant as a bridge or"step-stone" for learning the basic concepts of programming, without the distractions of a large language. However, it also converts into other languages (especially Java) very quickly and easily. As a result, it ultimately introduces both programming principles and practice in Java in a very short time.

This reference is a brief description of the language JJ. It is meant to be a definition and reference for beginners. However, it is not a tutorial on the language; it is not sufficient for beginners to learn programming. It concentrates mainly on the details of the language and does not deal with how best to use the language. It should be read over quickly now, so that you may know where to go when you have questions about the language. It could be read quickly also at various later times to serve as a review. The formal definition of the language in an abstract form called EBNF is also available elsewhere for programmers and professors. A diagrammatic definition is also given in the main text.

This reference describes the form (look, structure, layout, style, indentation, grammar, etc) of programs in JJ; it does not describe the process of programming, or the meaning (semantics) or the use (pragmatics) of the language.

Examples

Traditionally, the first example is to output "Hello world!"; in JJ the entire program is simply the following one line command (instruction or statement):

Output `"Hello world!"`

Incidentally, this same first example in Java would involve six new words (class, public, static, void, main, String, args) and three different kinds of brackets (rounded, square, and curly shapes). These concepts will be introduces in JJ, (and justified in Java) but at a later, more appropriate time

A short program follows to show the typical form (or syntax, layout) of a simple program in JJ. Programs involving classes and methods (functions and routines) are somewhat more complex, and discussed later.

The part to the right, consisting of line numbers and comments is not part of the language, and not part of the documentation; it just serves to describe what is happening in this case. You could guess at the meaning (or semantics).

```
-- Name Ann Onymous                          --1  indicates the name of the programmer
-- Does simple gross pay                     --2  documents the purpose of the program

   Boxes pay, hours ofType int               --3  declares 2 integer boxes pay & hours

   Outputln "Enter the hours"                --4  prompts the user to enter data
   Input hours                               --5  waits for input of data hours
   Output hours                              --6  echos the data entered

   If (hours <= 40) then                     --7  Checks if hours are less than or equal 40
      Set pay = 10 * hours                   --8  and if true sets pay to the product
   Else -- overtime                          --9  otherwise it computes overtime pay
      Set pay = 400 + 15 * (hours - 40)      --10 which is given by this formula
   EndIf                                     --11 end of the Choice construct

   Output "The gross pay is "                --12 prints everything within the quotes
   Outputln pay                              --13 prints the pay on the line above
                                             --14 and then prints a new line
```

Run, or execution of the program is as follows: first is a prompt for an input, and when a number such as 40 is entered the outputs is "The gross pay is 400". Similarily in a second run, when an input of 50 is entered the outputs is "The gross pay is 550".

Structure: Form of the Code

Program code is a mix of English, mathematics and logic. Lines in JJ begin with a capital letter, like sentences in English. In some languages (Java) a semicolon is used to end or terminate a command, and in other languages (Pascal) a semicolon simply separates commands, but in JJ semicolons are not needed. Two commands cannot be on the same line; but a command can extend over many lines.

Case-sensitivity is important; capital letters (upperCase characters) are considered very different from lower case letters. For example, a word like PAY all in upper case is not the same word as the word pay all in lower case or Pay in mixed case.

Spacing is significant. Note the gaps between some lines, which serves to separate some parts while at the same time unifing parts; for example lines 4, 5, and 6 all involve input and are separated from line 3 involving declaration and from line 7 involving a decision on the pay. Similarly lines 12 and 13 involve output.

Tokens are "clumps" or sequences of characters. Some tokens, consisting of decimal digits (0 to 9), represent numbers, and other clumps of characters represent words. Another token is the two character sequence "<=" which represents "less than or equal to". The two characters cannot have spaces between them; the two must be together. Tokens however should have spaces on either side of them.

Comments provide documentation or help for human readers. In JJ comments involve two dashes which make up the token "--". A comment consists of this token followed by any number of characters, up to the end of line. For example line 9 has a comment indicating overtime. Other languages have 2 or 3 different kinds of comments.

Programming deals with entities which are viewed as boxes (or variables) of a certain type (size) having values (contents) and given identifiers (names).

Keywords are words which are part of the language, and so are reused often. They are often called "reserved" words, and should not be used in any other ways in any program. They are listed below, in alphabetical order:

```
Acos, and, are, Asin. Atan, bool, BoolToStr, Box, Boxes, Call, Class, Constant,
Constructor, Cos, Dec, Else, ElseIf, EndClass, EndConstructor, EndFunction, EndIf,
EndRepeat, EndRoutine, equals, ExitOn, Exp, false, Floor, Function, If, Import, Inc,
IndexOf, Input, int, IntToReal, IntToStr, is, length, Log, New, NewArray, none,not,
null, ofClasss, ofType, or, Output, Outputln, pow, private, public, Random, real,
RealToInt, RealToStr, Repeat, Routine, Set, Sin, Slot, Sqrt, Start, Str, substring,
this, toLowerCase, toUpperCase, true, with.
```

Identifiers are names that programmers may choose for various objects and actions. Identifiers begin with any letter (a .. z, A .. Z) followed by any number of letters or digits (0 .. 9) or underscores "_". Identifiers are not limited in length. An underscore should not end an identifier. Examples of proper identifiers are:

```
pay, hours, i, over21, maxAge, min_Age
```

Data Types

JJ is a strongly typed language; all boxes (variables) must be declared along with their type. The basic (simple or primitive) data types are int, real, and bool. These correspond to int, double and boolean in Java.

Strings are similar to the basic types but come from a class Str. They consist of sequences of characters surrounded by quotes. Examples are:

```
"a",  "CA", "Hi mom", " "(123) 456-7890"
```

Characters are simple strings having only one item within them; they are of length 1.

Declarations bind (associate or fix) an identifier with a data type. They have the following forms:

```
Box   <identifier>  ofType  <type>
Box   <identifier>  ofClass <classType>

Boxes <identifiers> ofType  <type>
Boxes <identifiers> ofClass <classType>
```

Examples of some declarations follow; they must come before their use.

```
Box age ofType int
Boxes radius, area, diameter ofType real
Boxes tall, dark, handsome ofType bool
Boxes reply, grades ofClass Str
```

Mixing of different types is not possible in JJ. For example, reals cannot be added to ints; one of them must be converted. Conversion is not done automatically, but the conversion (or casting) can be done with functions such as IntToReal, RealToInt, BoolToStr, etc, described later.

Numeric types in JJ are only two; integer and real ; in Java there are about 6 integer types, and two real types (called double and float). Integers are used for counting, reals are used for measuring.

Integer is the type which describes whole numbers, both positive and negative. The type "int" in Jr refers to values whose maximum is about 2 billion (actually - 2,147,483,647 to 2,147,483,647, based on the binary; 2 raised to the power 31). Examples are:

```
1, 7, -13, 1984, 1234567890
```

Real numbers have a fractional part, and involve a decimal point. They are usually the result of some measurement (of size weight, rainfall, etc). Examples of real values are:

```
0.1, 2.0 -3.4, 3.14159,  98.6, 123456.789
1.23e4  (which is also 1234)
4.56e-5 (which is 0.0000456)
```

Logical boxes (of type bool) can have one of only two values: true, false. Operations of this type, in JJ, are three: `and, or, not`.

the result of (x and y) is true only when both x, y are true,
the result of (x or y) is true when either one or both x, y are true,
the result of (not x) is true when x is false; it is false when x is true.

Arithmetic operators on both ints and reals are the four familiar arithmetic functions, corresponding to addition, subtraction, multiplication and division, which have the symbols `(+, -, *, /)`.

Division of reals and ints differs. Division of reals results in a quotient which is a real. Division of ints results in a quotient which is an int. For example, division of the reals

```
9.0 / 5.0 results in 1.8
5.0 / 9.0 results in 0.5555555 ...
```

Division of ints, in the form x / y results in the quotient:

```
9 / 5 results in 1
5 / 9 results in 0
```

Mod, sometimes called modulus or rem (remainder) is another operator which also applies only to ints, and has the operator symbol %.

x%y results in the remainder when x divides y

For example

```
9 % 5 results in the remainder 4
5 % 9 results in the remainder 5
```

Arithmetic expressions, involve operators, values, boxes, and include parentheses. The binary operators of multiplication and division have precedence over addition and subtration. With operators of the same precedence they are associated from left to right. For example the expression (1 - 2 - 3) is evaluated as (1 - 2) -3 which is -4 rather than 1 - (2 - 3) which is 2.

Relational operators, as follows, compare two values:

```
<   stands for "less than"
>=  stands for "greater than or equal to"

<=  stands for "less than or equal to
>   stands for "greater than"

==  stands for "equal to"
!=  stands for "not equal to"
```

Notice that the lines are in pairs which are complements (opposite or negative).
For example, the complement of (x<y) is (x>=y);it is not (x>y).

Conditions are logical (boolean) expressions; they consist of proper combinations of relations, operators, and values.Some examples are:

```
(x != y)
(x <= y) and (y <= z)
(a == 7) or (a == 11)
```

Actions (commands or instructions) in JJ are many, but far fewer than in other languages. The simpler actions include Set, Input, Output, Outputln, Call; more complex actions, not considered here, include New, NewArray, Debug, Try, and Check.

Set, also called assignment, is of the form

```
Set <lhs> = <expression>
```

which replaces the current value of a box (indicated by lhs, leftHandSide> by the new value corresponding to the expression; both must be of the same type.

Boxes (or variables) are not automatically initialized; they must be set before they can be used.

Usually the left side is a variable. It cannot be of the form "obj.field"; JJ does not allow fields of an object to be changed directly, but must be done by a method. Java does not have this.

Input of a value to a box is of the form:

```
Input <boxName>
```

where there is a single box which may be of type int, real, bool or Str.

Output and Outputln (Output-line) are similar in form

```
Output <exp>
Outputln <exp>
```

The expression is evaluated and the value is printed.
However, Outputln also outputs a new line after it has printed.

Inc, Dec, are of the form:

```
Inc <intBox> by <intExpr>
Dec <intBox> by <intExpr>
```

where the value of the integer box is incremented or decremented by some integer value.
Both Inc and Dec apply only to integers, not reals.

Constructs: Choice and Loop

Programming constructs in JJ are two; the Choice construct (If) for selecting and the Loop construct (Repeat) for iterating. Both involve conditions and actions, in different ways.

Choice is a construct of the form:

```
If (expr) then
    action
{ElseIf
    action}
[Else
     action]
EndIf
```

where there may be any number of (ElseIf action) parts and zero or one (Else action) parts. The actions may be Set, Input, Output, Call, or another construct (If orRepeat).

The If, ElseIf, Else and EndIf must be indented the same amount (be in the same column). Ifs nested within other Ifs cannot begin in the same columns; deeper nested Ifs should be indented further.

An example of a Choice form follows:

```
-- Price depends on quantity
   If (quantity == 1) then
      Set price = 99
   ElseIf (quantity == 2) then
      Set price = 95
   ElseIf (quantity <= 4) then
      Set price = 90
   ElseIf (quantity < 6) then
      Set price = 85
   Else -- half dozen or more
      Set price = 75
   EndIf
```

Repeat is a looping construct of the form:

```
Repeat
  <action>
ExitOn <condition>
   <action>
EndRepeat
```

The Repeat, ExitOn and EndRepeat words must be alligned. If repeats are nested then the various ones must not be alligned with other Repeats.

An example of this Repeat looping construct follows:

```
-- Average many real values
-- ending with any negative terminator
   Set sum = 0.0 -- real value
   Set num = 0   -- int count

   Repeat
      Output "Enter value "
      Input value
   ExitOn (value < 0.0)
      Outputln value
      Set sum = sum + value
      Inc num by 1
   EndRepeat

   Output "The mean is "
   Outputln sum / IntToReal (num)
```

Methods (Functions and Routines)

Methods in JJ are of two very different forms (functions and routines). Both methods involve a name and slots (parameters or arguments).

Function methods are parts of expressions, and have a type; their names are usually nouns (maximum, average,). Routines are commands that stand on their own; their names are usually verbs (sort, spellOut, etc).

Calling (or invoking) functions differs from that of routines in that functions are parts of expressions as in:

```
   Set largest = maximum  (first, second)
```

whereas routines stand on their own as in

```
   Call spellOut with (digit)
```

Routines are defined in the following form, consisting of a head and a body which follows the Does part.

```
Routine <name> ( <slots> )
   <slotDeclarations.
   <local declarations>
   [<preCheck>]
-- Does <describe>
   [Start]
    <actions>
EndRoutine <name>
```

Heads of routines (or headers, or signatures) consist of a name and a bracketed list of slots, followed by a listing of the slots along with their type (in the same order as the above bracketed list). This is followed by a list of local boxes, and optionally by a preCheck. If there are no slots the word "none" is inserted between the brackets.

Bodies of a routine consists of a number of actions; one of the actions (only one per class) may be "Start"

Does, is a line which separates the head from the body; it indicates briefly the purpose of the method.

Names of routines at the head must match the name on the last line (tail) of the definition.

An example of a definition of a routine follows; outRow(times, mark) prints a row of marks a given number of times. For example, the command

```
Call outRow with (80, "*")
```

prints a row of 80 asterisks.

```
Routine outRow (times, mark)
   Slot times ofType  int
   Slot mark  ofClass Str
   Box count  ofType  int
-- Does draw a row of marks
   Set count = 0
   Repeat
   ExitOn (count == times)
      Output mark
      Inc count by 1
   EndRepeat
EndRoutine outRow
```

Functions are defined in the following form:

```
Function <name> ( <slots> ) ofType <type>
   <slotDeclarations>
   <result declaration>
   <local declarations>
   [<preCheck>]
-- Does <describle>
   <actions>
   result = <expr>
EndFunction <name>
```

This is very similar to the definition of a Routine except that the first line of the header must indicate the return type. Also there is a local variable result which must be declared as a local box.And the result must be assigned a value within the body.

An example of a function definition follows; it computes the area of a triangle given the lengths of the three sides.

```
Function triArea (a,b,c) ofType real
   Slot a ofType real -- three
   Slot b ofType real -- sides of
   Slot c ofType real -- triangle
   Box result ofType real
   Boxes s, t ofType real
-- Does use Hero's formula
   Set s = (a + b + c) / 2.0
   Set t = s*(s-a)*(s-b)*(s-c)
   Set result = Sqrt (t)
EndFunction triArea
```

Both kinds of methods when used within a class may have their visibility specified by appending to the first header either "is public" or "is private"

Intrinsic methods, are methods which are predefined or built into the language:

```
Sin(r)   returns the sine of an angle r, given as a real value in radians
Cos(r)   returns the cosine of an angle r, given as a real value in radians
Tan(r)   returns the tangent of an angle r, given as a real value in radians
Asin(r)  returns the arcsine of real value r, where r is in range [-1.0 to +1.0]
Acos(r)  returns the arccosine of real value r, in range [-1.0 to +1.0]
Atan(r)  returns the arctangent of real value r, where r goes from -Pi to Pi

Sqrt(r)  returns the square root of any positive real value
Pow(x,n) returns the value of real value x raised to real power n
Log(r)   returns the natural logarithm of real value r
Exp(r)   returns value of e raised to the r-th power
         where (e = 2.718, the base of the natural logarithm)
Ceil(r)  returns the smallest integer value which is not less than real value
Floor(r) returns the largest integer value less than or equal to r

Random() returns a real value between 0.0 and 1.0 (but not including 1.0)

IntToReal(i) returns the real value corresponding the the int value i
RealToInt(r) returns the integer value corresponding to the real value r

IntToStr(i)  returns a string corresponding to the integer value of r
RealToStr(r) returns a string corresponding to the real value of r
BoolToStr(b) returns a string corresponding to the boolean value of r

toLowerCase(s) returns the current string with all characters in lower case
toUpperCase(s) returns the current string with all characters in upper case

indexof(s) returns the index of the first occurrence of string s
           in the current string, but indicates -1 if there is no occurrence
substring(i,j) returns the substring of chars from i up to j
equals(s) returns true when the current string is equal to another string s
```

These intrinsic methods do not need to be imported from any class, but JJ can use methods which are imported from Java classes, such as the Math class.

Arrays in JJ are limited to a single-dimension; arrays of two dimensions and more dimensions can be created as classes from the single-dimensional arrays.
Arrays are accessed by integer indices within square parens [and] where the indices range from 0 to (n-1).

Declarations of arrays simply specify the type inthe form

```
Box   <name>  ofType  <type>[ ]
Boxes <names> ofType  <type>[ ]
Box   <name>  ofClass <class>[]
Boxes <names> ofClass <class>[]
```

for example:

```
Box hours ofType int[]
Boxes lastName ofClass  Str[]
```

Creation of arrays has one of the general forms

```
      NewArray <name> ofType  <type> [size]
      NewArray <name> ofClass <type> [size]
```

for example

```
      NewArray hours ofType real[7]
      NewArray grade ofClass Str[n] -- given n
```

Access of entries of arrays follows:

`Set a[i] = x --` puts value x into index i of an array a
`Set y = a[j] --` gets value from index j of an array a into y

Classes in JJ, at the simplest level, have the form:

```
Import <classLibrary>
Class <className>
-- Name: <username>
-- Does: <describe>
   <declaration of attribute Boxes with visibility>
   <constructor>
   <methods>
EndClass <className>
```

where

Constructors have the form:

```
Constructor <className> ( <slots> ) is <publicORprivate>
   <slot declarations>
   <actions>
EndConstructor <className>
```

For example, a bank account class is defined as follows:

```
Import JJIO
Class Account
-- Name Ann Onymous
-- Does provide a bank account

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

Constructor Account (b,n) is public
   Slot b ofType real
   Slot n ofClass Str
-- Does open the account
   Set balance = b
   Set name = n
EndConstructor Account

Routine credit (amount) is public
      Slot amount ofType real
-- Check amount is positive
   Set balance = balance + amount
EndRoutine credit
```

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

   Routine test (none) is public
      Box his ofClass Account
      Box her ofClass Account
   -- Does test Account class
   Start
      New his ofClass Account with (100, "Dad")
      Call his.credit with (50.00)
      If (his.covers (200.00))then
         Output "OK"
       EndIf
   EndRoutine test

End Class Account
```

Not covered in this reference are some advanced concepts involving design by contact, which include: Invariant, Check, PreCheck, PostCheck, Debug, TryCall, TrySet.
They will be considered in a sprecial section on "Programming by Contract".