JellyJ

Language Specification White Paper

Mohammad Mahdi Amirian
Jamal Pishvayee
Hojjat Sheikh Attar
Behrooz Nobakht

Shraif University of Technology
Computer Engineering Department

Preface

JellyJ has been started for the project of the compiler course. This projects is in her early stages and needs much effort to enhance. The current members are the attendants of the course. Thus, all opinions are warmly welcome for the project. The contact information following here can be used to honor us:

1. **JellyJ Design Group** — **jellyj@users.sf.net**
2. **Mohammad Mahdi Amirian** — **amirian@users.sf.net**
3. **Hojjat Sheikh Attar** — **attar@users.sf.net**
4. **Jamal Pishvayee** — **seyyedjamal@users.sf.net**
5. **Behrooz Nobakht** — **behrooznn@users.sf.net**

We are waiting for you.

0. A word with the Reader

In the text following, we have assumed that the reader is familiar with basic OO concepts. These concepts may consist of classes, objects, methods, attributes, inheritance, interfaces, etc. Moreover, this text is not to be used as a tutorial. Merely, it is introducing the JellyJ language and may be referenced as the language specification.

1. What is JellyJ?

It is a pretty, simple, pure object oriented programming language. Its main goal is to design an OO language without complexity of existing languages, which is divided into two major sub-purposes. This project is following two major aims in its development process. The one that is more significant is to design a simple "educational" programming language excluding the complexities and peculiarities of the common powerful PLs. The other one through this project is to simplify and purify a common OO PL like Java for the purpose of using in this project.

2. Why JellyJ?

On the other hand, JellyJ syntax is similar to Java to some extent. You have heard a lot about Object-Oriented programming. You have even tried many times to learn it (or even teach it to somebody) with SmallTalk, C++, Java, etc. but you were engulfed with the "too" strict syntactic restriction and got bored. You gave it up! Now, you can try ours. JellyJ is simple, Object-Oriented, pure and easy to learn. You will enjoy what is called Object-Oriented. The JellyJ syntax makes the OO principles seem natural and easy to catch for the novice programmer.

One of our main principles is conceptual consistency, which means reducing exceptions in language concepts. We have followed the way to reduce two specific rules into one more general one. We believe learning limited but pervasive rules is better than learning too many specific rules. Thus, it is tried best to design JellyJ so that as few rules as possible are used by the programmer to code.

Moreover, to assure consistency, a great attempt has been made to put only one solution for every single problem. In accordance with this principle, JellyJ will avoid any exceptions in its conceptual structure and design. Every rule is as general as possible without making exceptions to particular inconsistencies in the rules.

3. JellyJ Fundamentals

In JellyJ, all codes are embedded in classes and there is not any non-class-based code. Each class consists of attributes and methods. Both of class definition and declaration for each class are placed in one file. But each file can hold more than one class. In other words, a file is a package of classes and programmer name the package with a language structure at the first line of the file.

There is no distinction between classes and types. All types are classes and all classes are types. So we can use “type” and “class” interchangeably. There are two kinds of classes in the language: “Primitive Classes”, and “Regular Classes”, which will be described later in the text.

An instance of a class is called an Object. An Object can store data inside itself and it has some state in every instant of time.

Methods and attributes are also two kinds: “Class-Scope”, and “Object-Scope”. Class-Scope methods and attributes can only be used with identifying the class name and Object-Scope methods and attributes are referenced with the aid of an Object identifier. Class-Scope and Object-Scope will be discussed later.

3.1 Classes

3.1.1 Primitive Classes

Observing the world, you will find that there are two kinds of existents. Some of them are inventions that are made by human, and some of them are primitive existents that are not produced by human. Paying more attention to these two kinds, it will become crystal-clear that the human-made existents (inventions) are constructed from the primitive existents, and human can make instances of this type of existents, however, primitive existents are not made by human and they can never make instances of them. Even if human makes something identical to Oxygen, he has made an instance of *artificial* Oxygen, not the primitive Oxygen. Returning to our purpose and OO programming, we assumed that we have some primitives in the programming language world and they are existents, so they are classes, but they are not regular classes. The programmer *cannot* create (make instances of) them. However, their instances are available in the programming world, same as the primitives in the real world.

As an example, Integer is such a class; its instances (objects) are available in the programming world. “1” is an instance of Integer class; we have only one “1” there, so the programmer cannot make an instance of Integer class because all of Integer numbers are existing instances of Integer class. Thus, never do we need and permit to make an instance of Integer class or remove one of its instances.

String is another example of Primitive Classes. You can imagine the infinite set of string instances, all of which are available instances of the String class.

An example:

```
String name;
// Assigning some thing to name:
name = "JellyJ";
```

Note that in the above example we didn't make any instance of String class; "JellyJ" is an available instance of the class and we simply assigned it to "name".

JellyJ Primitive Classes are: **Integer**, **Real**, **String**, **Boolean**.
Their interfaces (operations) are described in appendix A.

An example:

```
// Defining a reference to Integer
Integer myInt;
// It is null after running this line

myInt = 23
// 23 is an available instance of Integer class, we assign myInt
// to a reference to this object.

myInt = myInt.add(3);
// It is similar to write 23.add(3)
```

3.1.2 Regular Classes

3.1.2.1 Class Structure

You, as the programmer, may define your own class (type). You may make instances of the class (i.e. create objects from the class). As mentioned before, a class consists of methods (operations and behaviors) and attributes (data). Thus, access level of attributes and methods should also be mentioned. Each attribute or method may be declared to be *public*, *protected* or *private* and the programmer can set the accessibility of the method or attribute with these keywords.

Note: *The **friendly** access level has been omitted from the language to reduce the language concepts as much as possible to follow the consistency rule of the OO concepts. **Friend** methods and attributes are not OO essentials; they are only for the programmer's convenience.*

A Regular class can be defined as below:

```
[class header] class identifier [inheritance]? [generic classes]?
{

[ [static]? [attribute declaration] | [method declaration] ]*

}
```

From here to end of the text:

> ***If we use a word between the brackets (i.e., [foo]), it means that the word itself is not a keyword, but this word points to a set of words or expressions that can be replaced with:***
>
> ***[something]?, [something], [something]+, [something]*:***

[something]?:
It is optional to replace [something] with the mentioned set. In other words, it denotes 0 or 1.

[something]:
It is mandatory to replace [something] and also you can only replace one member of the set. It denotes exact one.

[something]+:
It is mandatory to replace [something] and also you can put one or more members of the set. It denotes one or more than one.

[something]*:
It is optional to replace something there and also you can replace more than one member. It denotes 0 or more. It should be mentioned that separators may be different such as ‘,’ or ‘;’.

```
 [class header] class identifier [inheritance]? [generic classes]?
{
	class-scope {
		[ [attribute declaration] | [method declaration] ]*
	}

	object-scope {
		[ [attribute declaration] | [method declaration] ]*
	}
}
```

A regular class may have a constructor. It is a class-scope method that is called when an object is constructed. Also the programmer uses this method to construct an instance from some class. In JellyJ, the name of this method (constructor) is *makeInstance(…)* which may have some arguments .

An example:

```
// Declaring class shape
class Shape {}



...

// Defining myShape as a reference to the Shape class

Shape myShape;

// Making an instance:

myShape = Shape.makeInstance();
```

3.1.1.2 Class Header:

```
[class header] = abstract?
```

Sometimes, for future provision, the programmer does not want to implement a behavior (a method). The reason may be because of inheritance hierarchy or different usages and implementations of a specific behavior by distinguished users. The keyword **abstract** assures that the programmer may define a behavior without implementation of it. The restriction caused by this keyword is that the programmer is limited to create objects form classes for which there is no **abstract** unimplemented behavior.

3.1.1.2 Inheritance:

```
[inheritance] = inherits class-name implements interface-name+
```

3.1.1.3 Generic Classes

```
[generic classes] = identifies < class-name+ >
```

Usually, one of the common factors of the powerful programming languages is the capability of *type parameterization.* Type parameterization or *generic classes* permits the programmer to define general and useful behaviors, although he/she does not *specify* the creatures for which these behaviors will be adopted. As an example, take the very common behavior of *sorting*. It is very wise to define a general but useful behavior named *sort(arguments).* However, it is completely desirable not to specify the *arguments.* The reason is that:

(a) Considering every possible creature makes the code huge.
(b) For new types of creature a new code should be inserted.

However, the generic class allows the programmer to leave sort as general as possible. Though, he/she should tell the compiler that this *sort* is a generic class. When using *sort*, the programmer should specify the creature on which *sort* will takes place.

Unfortunately, in the original Java this feature has not been considered.

3.1.3 Interface Classes

```
interface class identifier [extends interface-class-name+]?
```

Very Often, even in the real world, there exist many behaviors that do not belong to specific creatures, however, different ones have different implementations of the behavior. We all think, but every human thinks differently. **Interface** classes introduce this feature, which is very useful according to OO concepts. It also should be mentioned that when a class is declared as an **interface**, it is the same as to declare that class as pure **abstract**.

3.2 Attributes

```
[attribute declaration]  =  const? class-name identifier;
```

In JellyJ, when an attribute is declared, it is assigned to the value of **null**. This approach comes from JellyJ design where the programmer is *prevented* from

assigning values to attributes when declaring them. The reason turns back to consistency concepts. *Attributes* should be initialized in the *constructor*. This rule has some advantages:

(a) Makes *every* initialization of *all* attributes to be placed in the *constructor*. This restriction makes the code more *readable* and *clear*.

(b) It makes the process of leaming *easier*.

Moreover, in JellyJ, there exists *no attribute,* which is declared **public.** This rule is placed in the design to conform to the Object Oriented and consistency concepts. However, usually, this rule has been disobeyed for the *convenience* of the programmer in many languages such as Java. However, we are not to be convenient in JellyJ.

3.3 Methods

3.3.1 Method Declaration

```
[method declaration] = final? overriding? abstract? [access modifier]
    class-name identifier([argument list]?) [throw statement]? ;

[argument list] = [const class-name identifier]*

[throw statement] = throws [class-name]+

[access modifier] = public | private | protected
```

Typically, when a class inherits another class, there are a number of choices to make:

(a) The subclass wants to add some completely new behaviors, which is simply direct It just adds the new ones.

(b) The subclass needs to modify one of the behaviors of the Superclass. This action is called **overriding**. The subclass **override** s one of the methods of the superclass.

Sometimes, the superclass determines that some specific behavior that has been implemented by it does not need anything more. In this case, the superclass defines this special method as **final**. In this way, none of his children may *never* **override** this method. (Just the same as the story of the classes of type **barren**).

When a method is defined as **abstract**, it means that the implementation of the behavior will come later in the class hierarchy or when somebody wants to use this class. Refer to **abstract** described ahead in the text.

3.3.2 Special Methods

3.3.2.1 Constructor

```
public classname makeInstance(argument list);


classname.makeInstance();
```

In JellyJ, *every* class has a *class-scope* method, which has the responsibility of:

(a) Making an instance of the class.

(b) Initializing the attributes provided that they are specified in this method.

This special method is called *constructor*. Constructor is an optional feature. However, when there is no constructor defined by the programmer, JellyJ compiler will define a *default* constructor, which only makes an instance of the class. A class constructor is not **inherit**ed. The reason is that every single class has attributes that are **private** by default, so a subclass may never access attributes of the superclass. Though, there have been some provisions that a subclass can use the super class's constructor. Refer to Alias section.

An Example:

```
// First, we declare a class representing a person. Each Person
// has a 'name'. This name would be given to him once when he/she //
is born.
  class Person {
     class-scope {
        public Person makeInstance (String pname) {
         name = pname;
        }
     }
     object-scope {
       private Sting name;
     }
  }
// And, now we can use the Person to give birth to a person named
// "JellyJ".
  ...
  Person p;
  p = Person.makeInstance ("JellyJ");
  ...
```

Notes on Constructor method:

- One can overload **makeInstance** to implement various initialization.
- If a class does not have any *written*, the constructor for it will have a default with no argument.
- Constructors will not be inherited.
- If the superclass does not have a *default* constructor, the subclass must have at least one constructor and, in every constructor it has, it must call one of the Superclass constructors in the first line.

```
An Example:
  class Subclass inherits Superclass{
    class-scope {
      public Subclass makeInstance(String param){
         Superclass.makeInstance(param);
         ...
      }
    }
```

```
    ...
}
```

3.3.2.2 Static Initialization

Class-scope attributes must have some *value* when defined in the class for the first time. Accordingly, one can write special a method with a standard name **installClass** and add the initializations in the method. This method will be automatically called and its signature must be as illustrated below:

```
class Subclass {
  class-scope {
    public void installClass() {
       objectCounter = 0;
    }
  }
  ...
}
```

3.4 Aliases

All data are objects in this language and all operations are done via method calls. This is nice, but there are some problems. For example, we introduced add method for Integer Primitive Class to add to Integer numbers. So we should write the following line to add 49 to 123:

```
Integer result;
Result = 123.add(49);
```

However, programmers prefer to use '+' operator to add to numbers and it is almost a standard. What must we do in this case ? Accepting Integer Class as an *exception* (provide operator overloading for it)? Though, we chose another way, **aliases**!

We prefer to accept Integer Class as an exception only in *syntax* not in *semantic*. It means that we provide a way for some of the classes, which need operators or any non-Object-Oriented syntax to allow users to write some non-pure object-oriented code. Thus, the programmer uses this syntax as aliases and compiler will replace them with the regular method calls. The programmer must know that this syntax is added only for *convenience*. So, only a finite number of aliases are available in the language and programmer can't add or remove anything to/from them.
Aliases example:

b1 || b2 (b1 and b2 are both Booleans) → b1.or(b2)
i1 – i2 (i1 and i2 are both Integers) → i1.sub(i2)
-j (j is an Integer or a real) → j.neg()
1[3] (l is an array) →

The complete list of aliases will be in Appendix B.

3.5 Statements

3.5.1 Variable Declaration

```
const? class-name identifier [ = expression]?
```

3.5.2 Control Structures

All control structures in JellyJ are the same as Java.

3.5.2.1 If Structure:

```
if ([condition])
      statement
```

3.5.2.2 For Structure:

```
for ([initialization]*; [condition]; [update]*)
      statement
```

3.5.2.3 While Structure:

```
while ([condition])
      statement
```

3.5.2.4 Do-While Structure:

```
do
      statement;
while ([condition])
```

Note:

In the mentioned structures, if more than one statement is required, they are grouped in a block with "{ }", and ";" is not needed after "}".

3.5.2.5 Switch Structure:

```
switch (identifier) {
[case value: [statement;]*]+
[default : [statement;]*]?
}
```

3.5.2.6 Try-Catch-Finally Structure:

```
try {
      [statement;]*
}[catch (exception-type identifier} {[statement;]*}
]+
[finally {[statement;]*}]?
```

3.5.3 Assignments

We have two types of assignment.

```
identifier = expression

identifier ?= expression
```

One of the most frequent problems in Object-Oriented programming is the case of *casting objects*. It is very often and useful that a

programmer wants to cast (up or down) two related objects. In the case of *upcasting*, there usually will not raise a problem. On the other hand, when using *downcasting,* there is always some cost to check whether the cast is appropriate or not. In JellyJ, we have solved this problem with the aid of two nice concepts:

(a) Generic classes which have been described before.

(b) A new statement, which is called *assignment attempt.* In this concept, JellyJ compiler tries to do the assignment. Along this process comes also the concept of *casting.* If JellyJ compiler succeeds, it means that it was either a *simple assignment* or it has been a *cast* which has been qualifie d to be proper and correct. Otherwise, the compiler will return **null**. So, it has been an illegal cast.

Using assignment attempt will remove the difficulties of the process of casting.

3.6 JellyJ Rules

3.6.1 Class-scope methods and attributes

Against what is seen in Java , one can access class-scope attributes and methods only through the class name and can not access them through object instantiated.

An Example:

```
class Person {
   class-scope {
     private Integer count;
     public Integer getPersonCount(){
       return count;
     }
   }
    ...
}


..
  Person p = Person.makeInstance();
  p.getPersonCount();                 // ERROR
  Person.getPersonCount();            // Proper Use
..
```

3.6.2 Pre-defined Identifiers

Some words are pre-defined in classes to provide access to some essential items

- *This* is a reference to the object and is defined in object-scope methods.
- Super is a reference to the superclass object which will be instantiated.

3.7 Built-In Classes

3.7.1 Debug

One of the most useful assets to a programming language is the library of *Debug*. Usually this library contains routines for *assertion feature, error tracing, etc.* The Debug class facilitate the capability of the programmer to test, debug and trace his programmer.

Moreover, the *invariant feature* is also placed in the Debug class. This feature is used to assure certain conditions in the a class or a method. An invariant consists of some Boolean expressions containing the know arguments for a class or a method. This Boolean expression is always checked to be true on certain checkpoints.

The Debug class is not an essential to the programming language. But it will be considered as a complement library for it.

3.7.2 Compiler

In this library, usually, some auxiliary methods that are used in the context of objects and classes. These methods are usually used to gain extra information about objects and classes, such as isInstanceOf(), getClass(), etc.

In order to implement such library we need a class named *Class*. This class will represent a class that can store information of class types.

3.7.3 RunTime

3.7.4 I/O

One the important factors in qualification of the programming languages is the simplicity of I/O library.

It is obvious that a consistent and complete I/O hierarchy is an essential need. This factor has been completely conformed in Java. However, take a novice programmer, he/she wants to test the programming language with a simple “Hello World!” message. In Java, he/she will be puzzled up in the I/O hierarchy. In JellyJ design, we decided to simplify the rules while complying to OO hierarchy of Java I/O.

Appendix A

In JellyJ, it is to have only objects as creatures of the language. Therefore we need to declare some of the *primitive* operations used for the primitive classes. Also, there will be an equivalent *alias* for each of these operations. These operations are illustrated here in appendix A, and the aliases will be described in appendix B.

Integer Class:

```
class interface Integer
{

 class-scope {

      public static Real    toReal(Integer someInt) {…} ;
      public static String  toString(Integer someInt) {…};
      public static Integer maxInt(){…} ;
      public static Integer minInt(){…};
 }

object-scope {
      public Real    add(Real someReal) {…};
      public Integer add(Integer otherInt) {…} ;
      public Integer subtract(Real otherInt) {…} ;
      public Integer subtract(Integer otherInt) {…} ;
      public Integer multiple(Real otherInt) {…};
      public Integer multiple(Integer otherInt) {…};
      public Integer divide(Real otherInt) {…};
      public Integer divide(Integer otherInt) {…};

      public Integer mod(Integer otherInt) {…} ;
      public Integer negative(Integer otherInt) {…};
      public Integer power(Integer otherInt) {…};

      public Integer bitAND(Integer otherInt) {…};
      public Integer bitOR(Integer otherInt) {…};
      public Integer bitNOT(Integer otherInt) {…};

      public Boolean isGrater(Integer otherInt) {…};
      public Boolean isLess(Integer otherInt) {…};
      public Boolean isEqual(Integer otherInt) {…};
 }
}
```

Real Class:

```
class interface Real
{
 class-scope {

      public static Integer toInteger(Real someReal);
      public static String  toString(Real someReal);
      public static Real    toRealPart(Real someReal);
      public static Integer maxReal();
      public static Integer minReal();
 }
 object-scope {

      public Real    add(Integer otherInteger);
      public Real    add(Real otherReal);
      public Real    subtract(Integer otherReal);
      public Real    subtract(Real otherReal);
      public Real    multiple(Integer otherReal);
      public Real    multiple(Real otherReal);
      public Real    divide(Integer otherReal);
      public Real    divide(Real otherReal);
      public Real    negative(Real otherReal);

      public Boolean isGrater(Real otherReal);
      public Boolean isLess(Real otherReal);
      public Boolean isEqual(Real otherReal);
 }
}
```

Boolean Class:

```
class interface Boolean
{

// object scope methods:

      public Boolean and(Boolean otherBool);
      public Boolean or(Boolean otherBool);
      public Boolean not(Boolean otherBool);


// class scope methods:

      public static String toString(Boolean someBool);
}
```

String Class:

```
class interface Boolean
{

// object scope methods:

      public String  concatenate(String otherStr);
      public String  subStr(Integer from, Integer length);
      public Integer findStr(String otherStr);
      public String  putStr(Integer from, String otherStr);
      public String  rmStr(Integer from, Integer length);
      public Integer getLength();
      public String  lowerCase();
      public String  upperCase();

      public Boolean isGrater(String otherStr);
      public Boolean isLess(String otherStr);
      public Boolean isEqual(String otherStr);


// class scope methods:

      public static String toString(Boolean someBool);
}
```

Appendix B

In this section, aliases for the operations of primitive classes are included.

Alias	*Original*	*Usage Case*
n + m	n.add(m)	Integer, Real
n – m	n.subtract(m)	Integer, Real

– n	n.negative()	Integer, Real
n * m	n.multiple(m)	Integer, Real
n / m	n.divide(m)	Integer, Real
n % m	n.mod(m)	Integer
a > b	a.isGreater(b)	Integer, Real, String
a < b	a.isLess (b)	Integer, Real, String
a >= b	a.isGreat(b) | a.isEqual(b)	Integer, Real, String
a <= b	a.isLess(b) | a.isEqual(b)	Integer, Real, String
!a	a.not()	Boolean
a | b	a.or(b)	Boolean
a & b	a.and(b)	Boolean
s[i]	s.subStr(i,1)	String
s1 + s2	s1.concatenate(s2)	String