

# The Krakatoa Language

José de Oliveira Guimarães  
Departamento de Computação  
UFSCar - São Carlos, SP  
Brasil  
e-mail: jose@dc.ufscar.br

August 12, 2015

The Krakatoa language is a subset of Java with a few additions/modifications. Krakatoa supports all the basic concepts of object-oriented programming such as classes, inheritance, and polymorphism. Unlike Java, a program is composed by a single source file. Every class can only refer to the classes that appear textually before it in this file.

An example of a program in Krakatoa is shown below

```
class Store {
    private int n;
    public int get() {
        return this.n;
    }
    public void set( int n ) {
        this.n = n;
    }
}

class Program {
    public void run() {
        Store s;
        int a;
        s = new Store();
        read(a);
        s.set(a);
        write( s.get() );
    }
}
```

Every program must have a class named **Program** with a parameterless method called **run**. To start the execution of a program, the runtime system creates an object of class **Program** and sends to it a message **run**.

In the following sections we define the main elements of the Krakatoa language.

## 1 Classes

A class has the format

```
class Name {
    public and private members
}
```

in which **Name** is an identifier, the class name, that should be different from all previously declared classes. A member of a class may be a private variable or a public or private method. The next example shows the declaration of a class with private instance variables **x** and **ok** and public methods **sum** and **setOk**. Note the words “private” or “public” precedes each declaration. A class may have zero members. The **run** method of class **Program** must be public and it should not take any parameters.

```
class A {
    private int x;
    public int sum( int y ) {
        return this.x + y;
    }
    private boolean ok;
    public void setOk( boolean ok ) {
        this.ok = ok;
    }
    ...
}
```

The syntax for method declaration is

```
public ReturnType methodName( parameter-list ) {
    Statement-List
}
```

The return type may be a class, a basic type (**int**, **boolean**, or **String**), or **void** (no return value). The syntax for local variable declaration is equal to Java and described in the next section. The name of a method or instance variable should be different from the name of the previous members of the class. A member of a class is an instance variable or method. Statements include declaration of variables. So,

```
    int x, y;
counts as a statement.
```

A static variable is declared inside a class as

```
static private Type variableName;
```

A static variable can only be accessed by a static method, which is made through the syntax

```
    ClassName.variableName
```

A static variable must be private and it can have the same name as an instance variable of the same class.

A method may be declared static:

```
static public ReturnType methodName( parameter-list ) {  
    Statement-List  
}
```

In a class, no two static methods may have the same name. Different semantic rules apply for static methods, as in Java:

- a static method is called using the class name, as in “`Clock.getCurrentTime()`” in which `Clock` is the class name. This syntax must be used even when this call is inside class `Clock`. A static method can only call a previously declared static method;
- static methods can be public or private;
- keyword `this` cannot be used (see Section 1.3) inside a static method. Therefore neither non-static methods of the current class can be called inside a static method nor instance variables can be accessed. Of course, static methods can be called and message sends to variables and parameters are legal;
- keyword `super` (see Section 1.1) cannot be used inside a static method;
- regular, non-static methods, can call static methods;
- a static method cannot have the same name as another static method of the same class. But the name may be equal to the name of a non-static method;
- static methods of classes in a inheritance hierarchy are completely independent of each other. That means that if B inherits from class A, a static method of class B may have the same name as a class-A method;
- a private static method can be called by any other static or non-static method of the class in which it was declared.

Variables and parameters whose types are classes are in fact pointers, as in Java. Therefore, in

...

```
Store s, t;  
s = new Store();  
t = s;
```

the declaration of `s` and `t` does not create objects of class `Store`. An object of this class should be dynamically created with `new`:

```
s = new Store();
```

Memory deallocation is made by the garbage collector. Class `String` is much like a class. However, it is not possible to create a `String` object with `new`.

There is a global value `null` that can be assigned to any variable whose type is a class. `null` represents a value of a class that does not have methods.

## 1.1 Inheritance

Inheritance of a class A by a class B is made as

```
class B extends A {  
    ...  
}
```

Class B inherits all instance variables and methods of A. A public method of A may be redefined in B if its

signature (name, return value type, and parameter types) is not changed. The redefined method should also be public. It is allowed to redefine in **B** a private method of **A**. It is as if **B** defined a new method since no one knows the private methods of **A**.

Class **B** has no access to private instance variables and methods of **A**.

The statement

```
super.m(p1, p2, ... pn)
```

inside a **B** method orders the compiler to look for a **m** method starting at the superclass of **B**, **A**. If it is not found in **A**, the search continues in the superclass of **A** and so on. The parameters **p<sub>1</sub>**, **p<sub>2</sub>**, ... **p<sub>n</sub>** must be convertible (section 2.5) to the formal parameters to the **m** method found in the search.

Message sends to **super** are linked at compile time to a specific method of a superclass. No search for a method is made at runtime. Note that **super** cannot be used in static methods.

A final class is declared by using the keyword “**final**” before “**class**” as in

```
final class Earth {  
    ..  
}
```

A final class cannot be inherited. All message sends to its methods can be static. That is, there is no need of a search for a method at runtime.

A “**final**” method, declared as

```
final public void get() { ... }
```

cannot be redefined in a subclass. Only public methods can be final and a final class cannot declare a “**final**” method (since the methods of a final class are already final). No other restrictions apply. A class **C** may extend a class **B** that extends a class **A** that defines a method **m** redefined in **B** and **C**. The **m** method of **C** may be declared final.

## 1.2 Message Sends

The statement

```
s.calc(b)
```

is the sending of the message “**calc(b)**” to the object pointed to by **s**. This message send is valid if:

1. the declared class of **s** or one of its superclasses has a public method with name **calc**.
2. this method **calc** takes a sole parameter and the type of **b** can be convertible (see section 2.5) to the type of the formal parameter of **calc**.

At runtime, the runtime system (RTS) makes a search for a method **calc** in the class of the object referred by **s**. If it is not found there, the search continues in the superclass of the class of this object, the superclass of the superclass, and so on. When a method is found, it is called. The RTS will always find an adequate method except when the variable points to **null**.

A method that does not return a value can be used as a statement:

```
stack.print();
```

A method that returns a value must be called only within an expression as in

```
if ( stack.getSize() > 0 ) insert(0);
```

## 1.3 this

The keyword **this** represents a variable whose type is the class in which it is being used. **this** points to the object that received the message that caused the execution of the method. The same as in Java. Using class **Store** (first example), the code

```
s = Store.new();  
s.set(12);
```

causes the execution of method `set` of `Store`. Inside this method, `this` points to the same object as `s`. This keyword is used to handle the instance variables and to call private and public method of the class — observe class `Store`. The value of `this` cannot be modified by an assignment.

In a message `send this.m()` inside a class `A`, the compiler searches for a private method `m`. If none is found, the search continues in the public methods of `A`. If none is found, the search continues in the public methods of the superclass of `A`, the superclass of the superclass, and so on.

## 2 Basic Krakatoa Elements

### 2.1 Comments

Comments in the language are put between “`/*`” and “`*/`” and between `//` and the end of the line. Nested comments are not allowed, as in

```
/* comment /* another comment */ end of the first comment */
```

The comment ends in the first `*/` found. Note that “`/*`” and “`*/`” may appear inside a comment started with `//` — they will not mean a comment. The converse is also true.

### 2.2 Basic Literals and Types

There are only three basic types in Krakatoa: `int`, `boolean`, and `String`. Literals of type `int` must be between 0 and 32767. Any number of zeros before the number is allowed. Therefore the numbers

```
000000000000001
000000000000000
```

are legal. Type `boolean` has only two values: `false` and `true`.

`String` literals must appear between quotes: “`Do you always come here?`”

The backslash `\` can be used to remove the meaning of “” and the backslash itself. In fact, the string “`\c`” has the same meaning as in C regardless of the character `c`, which can be anything.

The comparison operators `<`, `<=`, `>`, `>=` can only be applied to `int` values. The comparison operators `==` and `!=` can be applied to `int` and `boolean` values.

The operators `==` and `!=` can also be used to compare expressions whose types are classes. The result is a `boolean` value. In a comparison `left == right` or `left != right`, if `LeftType` is the declared type of `left` and `RightType` is the declared type of `right`, one of three things must occur:

1. `LeftType` is `RightType`;
2. `LeftType` is convertible to `RightType`. See definition of *convertible* in section 2.5;
3. `RightType` is convertible to `LeftType`.

That is, classes `LeftType` and `RightType` must be related by inheritance. All other possibilities are illegal, for we know the expression `left == right` will evaluate to `false` anyway.

Operators `==` and `!=` can also be used when `LeftType` (`RightType`) is `String`. In this case, `RightType` (`LeftType`) must be `String` or `right` (`left`) must be `null`.

Operators `+`, `*`, `-`, and `/` apply to `int` values resulting in `int` values. The semantics of these operators is the same as in C.

The binary operators `&&` and `||` and the unary `!` accept `boolean` operands and have the usual meaning. The evaluation of the expression

```
left && right
```

start in `left`. If `left` is `false`, all the expression is considered `false`, even without the evaluation of `right` (which can be an expression). If `left` is `true`, `right` will be evaluated.

The expression

```
left || right
```

is `true` if `left` is `true`. If `left` is `false`, the result has the value of `right`.

The type of a variable, parameter, or return value of a method must be a basic type (`int`, `boolean`, or `String`) or a previously declared class, which may be the current class.

## 2.3 Identifiers

Identifiers are composed by letters, digits, and underscore (`_`), starting with a letter. Here are some examples of valid identifiers:

```
getNum  x0  y1  get_Num
```

The identifiers

```
_main  3ab  get$Num  write
```

are illegal. “`write`” is illegal because it is a keyword. The list of keywords of Krakatoa is

<code>boolean</code>	<code>break</code>	<code>class</code>	<code>else</code>
<code>extends</code>	<code>false</code>	<code>if</code>	<code>int</code>
<code>new</code>	<code>null</code>	<code>private</code>	<code>public</code>
<code>read</code>	<code>return</code>	<code>static</code>	<code>String</code>
<code>super</code>	<code>this</code>	<code>true</code>	<code>void</code>
<code>while</code>	<code>write</code>	<code>final</code>	

There is no limit in the number of characters of an identifier. Upper and lower case letters are considered different. All identifiers must be declared before used. None can be declared twice in the same scope (see below).

## 2.4 Scope

The scope of a class name is from the place it appears to the end of the file. The scope of an instance variable or method is from the place it is declared to the end of the class. However, instance variables and methods of the object that received the message must be handled using the keyword “`this`”:

```
this.n = 0;
x = this.m(5);
this.p.set(0);
```

Therefore a code like

```
set(0);
```

is always illegal because there is no receiver to the message “`set(0)`”.

The scope of local variables and parameters of a method is from the place of declaration to the end of the method body. A local variable cannot have the same name as a parameter of the same method. Local variables and instance variables are always distinguished because instance variables are accessed using `this`. Local variables have precedence over class names, which are global.

## 2.5 Assignment

The assignment of an expression `rightExpr` to a variable `left` is made as

```
left = rightExpr;
```

If `LeftType` is the declared type of `left` and `RightType` the type of `rightExpr`, this statement is valid if:

- `LeftType` is a basic type (`int`, `boolean`, or `String`) and `LeftType` is `RightType`;
- `RightType` is a class which is subclass of `LeftType`. We consider that a class is subclass of itself;
- `LeftType` is a class and `rightExpr` is value `null`.

If any of the items is valid, we say that `RightType` is convertible to `LeftType`.

The same rules apply to parameter passing to methods and return value by means of the command `return`. That is, statements like

```
bb.m(pr);
return r;
```

are valid if assignments

```

    pf = pr;
    x = r;

```

are valid, in which `pf` has the same type as the formal parameter of method `m` and `x` is a variable with the same type as the return value type of the method in which this statement is.

## 2.6 Decision Statement

The `if` statement has the form

```

if ( expr )
    Statement
else
    Statement

```

The part

```

else
    Statement

```

is optional. `expr` must be a `boolean` expression.

## 2.7 Repetition Statements

The statement

```

while ( expr )
    Statement;

```

means that `Statement` is to be executed while the `boolean` expression `expr` evaluates to `true`. This loop may be ended by executing statement `break`. Of course, it is illegal to use `break` outside a `while` statement.

## 2.8 Method Return Value

A method returns a value with the command

```

return expr;

```

If the type of the return value `expr` is `U` and the declared return type of the method is `T`, then `U` must be convertible to `T` (see section 2.5). That is, an assignment

```

t = u;

```

should be legal, in which the types of `t` and `u` are `T` and `U`, respectively. A method without a return value type cannot have a `return` command. A method with return value should have at least one `return` command.

## 2.9 Input and Output

Input is made by the statement `read`:

```

read( IdList );

```

where `IdList` is a list of one or more variables (local, parameter, static or instance) of the type `int` or `String`. The statement

```

read( a1, a2, ... an )

```

is equivalent to

```

read(a1);
read(a2);
...
read(an);

```

in which `read(a)` is equivalent to the following language-C code:

```

{ char __s[512];

```

```

    gets(__s);
    sscanf(__s, "%d", &a); }

```

if `a` has type `int`. If the type of `a` is `String`, `read(a)` is equivalent to

```

{
    char __s[512];
    gets(__s);
    _a = malloc(strlen(__s) + 1);
    strcpy(_a, __s);
}

```

in C.

Each variable is read in a separate line in the standard input.

The statement

```
write( expr1, expr2, ... exprn )
```

writes the expressions in the standard output. It is equivalent to

```

write(expr1);
write(expr2);
...
write(exprn);

```

The number `n` should be greater than zero. The statement `write(expr)` is equivalent to the following code in C if `expr` has type `int`:

```
printf("%d ", expr);
```

If the type of `expr` is `String`, this statement is equivalent to

```
puts(expr);
```

Boolean expressions cannot be parameters to `write`. Note that `write` uses exactly one space after the printed number.

Statement `writeln(expr)` is equivalent to `write(expr)` followed by `write("\n")`.

### 3 Metaobjects

Krakatoa supports a very limited version of metaobjects. A metaobject is an object that controls other objects. In this case, a metaobject is an object that controls the compilation process. We have no space here to describe them completely so we will just explain how the available metaobjects works.

Metaobject `ce` is a metaobject that communicates to the compiler that there is a compiler error in the source code. It takes from two to four parameters (the last two ones are optional). The first one is always the number of the line of the current source code that has the error. The second one is a string with a description of the error. The third one is the error that the compiler should sign. This is a suggestion — the compiler implementer is free to use other error messages. The fourth parameter is the method of the Krakatoa compiler that should be modified in order to make the compiler sign this error.

```
@ce(4, "O nome da classe está ausente", "Missing identifier",
    "comp.Compiler.classDec()")
```

```

class {
    public void run() { }
}

```

In this example, the metaobject call `@ce(...)` informs the compiler that there is an error in line 4. The error description is

```
"O nome da classe está ausente"
```

The compiler should issue an error message that is something like



"Missing identifier"

To make the Krakatoa compiler to sign this error, you should change method  
comp.Compiler.classDec()

After compiling the source code of this example, the Krakatoa compiler will issue a warning if it did not sign an error message. It should. If it signed, it will show the actual error message and the error message "Missing identifier". Then the user can compare if the compiler really signed the error with the correct error message. The Krakatoa compiler will also check if the line number of the error signed by itself is 4.

There is just one more metaobject: `nce`. It does not take parameters and informs the compiler that there should be no compilation errors in the source code. If there is any, the Krakatoa compiler will point out that it has a flaw.

@nce

```
class Program {  
    public void run() { }  
}
```

## 4 The Krakatoa Grammar

This section defines the language grammar. The reserved words and language symbols are shown between “ and ”. A sequence of symbols between { and } can be repeated zero or more times and a sequence of symbols between [ and ] is optional. Parentheses group symbols. For example,

$D ::= (A|B) \{ C \}$

means A or B followed by any number of C's.

The non-terminal **StringValue** represents a string of characters between quotes as in Java: "Hi !!". This non-terminal is **not** in the grammar.

The initial grammar non-terminal is Program.

AssignExprLocalDec	::= Expression [ “=” Expression ]   LocalDec
BasicType	::= “void”   “int”   “boolean”   “String”
BasicValue	::= IntValue   BooleanValue   StringValue
BooleanValue	::= “true”   “false”
ClassDec	::= “class” Id [ “extends” Id ] “{” MemberList “}”
CompStatement	::= “{” { Statement } “}”
Digit	::= “0”   ...   “9”
Expression	::= SimpleExpression [ Relation SimpleExpression ]
ExpressionList	::= Expression { “,” Expression }
Factor	::= BasicValue   “(” Expression “)”   “!” Factor   “null”   ObjectCreation   PrimaryExpr
FormalParamDec	::= ParamDec { “,” ParamDec }
HighOperator	::= “*”   “/”   “&&”
Id	::= Letter { Letter   Digit   “_” }
IdList	::= Id { “,” Id }
IfStat	::= “if” “(” Expression “)” Statement [ “else” Statement ]
InstVarDec	::= Type IdList “,”

IntValue	::= Digit { Digit }
LeftValue	::= [ ( "this"   Id ) "." ] Id
Letter	::= "A"   ...   "Z"   "a"   ...   "z"
LocalDec	::= Type IdList ","
LowOperator	::= "+"   "-"   "  "
MemberList	::= { Qualifier Member }
Member	::= InstVarDec   MethodDec
MethodDec	::= Type Id "(" [ FormalParamDec ] ")" "{" StatementList "}"
MOCall	::= "@" Id [ "(" { MOParam } ")" ]
MOParam	::= IntValue   StringValue   Id
ObjectCreation	::= "new" Id "(" ")"
ParamDec	::= Type Id
Program	::= { MOCall } ClassDec { ClassDec }
Qualifier	::= [ "final" ] [ "static" ] ( "private"   "public" )
ReadStat	::= "read" "(" LeftValue { "," LeftValue } ")"
PrimaryExpr	::= "super" "." Id "(" [ ExpressionList ] ")"   Id   Id "." Id   Id "." Id "(" [ ExpressionList ] ")"   Id "." Id "." Id "(" [ ExpressionList ] ")"   "this"   "this" "." Id   "this" "." Id "(" [ ExpressionList ] ")"   "this" "." Id "." Id "(" [ ExpressionList ] ")"
Relation	::= "=="   "<"   ">"   "<="   ">="   "!="
ReturnStat	::= "return" Expression
RightValue	::= "this" [ "." Id ]   Id [ "." Id ]
Signal	::= "+"   "-"
SignalFactor	::= [ Signal ] Factor
SimpleExpression	::= Term { LowOperator Term }
Statement	::= AssignExprLocalDec ";"   IfStat   WhileStat   ReturnStat ";"   ReadStat ";"   WriteStat ";"   "break" ";"   ";"   CompStatement
StatementList	::= { Statement }
Term	::= SignalFactor { HighOperator SignalFactor }
Type	::= BasicType   Id
WriteStat	::= "write" "(" ExpressionList ")"
WhileStat	::= "while" "(" Expression ")" Statement

## References

- [1] Stroustrup, Bjarne. *The C++ Programming Language*. Second Edition, Addison-Wesley, 1991.
- [2] Lippman, Stanley B. *C++ Primer*. Addison-Wesley, 1991.
- [3] Guimarães, José de Oliveira. *The Green Language*. <http://http://www.cyan-lang.org/jose/green/green.htm>.