

**Definition of DJ (Diminished Java)**  
version 1.0                      Jay Ligatti

## 1 Introduction

DJ is a small programming language similar to Java. DJ has been designed to try to satisfy two opposing goals:

1. DJ is a *complete* object-oriented programming language (OOP):
  - (a) DJ includes all the core features of OOPs like Java, and
  - (b) you can express any algorithm in DJ (more precisely, DJ is *Turing complete*; any Turing machine can be encoded as a DJ program).
2. DJ is *simple*, with *only* core features included. DJ can therefore be *compiled straightforwardly*; we can design and implement a working (non-optimizing but otherwise complete) DJ compiler in one semester.

## 2 An Introductory Example

Here is a valid DJ program:

```
// This DJ program outputs the sum 1 + 2 + ... + 100
class Summer extends Object {

    // This method returns the sum 0 + 1 + .. + n
    nat sum(nat n) {
        nat toReturn;
        // note: nat variables automatically get initialized to 0

        for(0; n>0; n=n-1) { toReturn = toReturn + n; };

        toReturn;
    }
}

main {
    // create a new object of type Summer
    Summer s;
    s = new Summer();

    // print the sum 0 + 1 + ... + 100
    printNat( s.sum(100) );
}
```

Many additional examples of valid and invalid DJ programs are posted at: <http://www.cse.usf.edu/~ligatti/compilers/22/a1/dj/>.

### 3 Format of DJ Programs

A DJ program must be contained in a single file that begins with a (possibly empty) sequence of *class declarations* and then must have a *main block*.

A *class declaration* consists of the *class* keyword, followed by a class name, then the *extends* keyword, then a superclass's name, then an open brace '{', then a (possibly empty) sequence of *variable declarations*, then a (possibly empty) sequence of *method declarations*, and then a closing brace '}'.

A *variable declaration* consists of a *type name* (either *nat* for a natural number, or a class name for an object type) followed by a variable name followed by a semicolon. For example, *nat i;* declares a variable *i* of type *nat*.

A *method declaration* consists of a return type name, followed by a method name, then a left parenthesis '(', then a parameter type name, then a parameter name, then a right parenthesis ')', and then a *variable-expression block*.

A *variable-expression block* consists of an open brace '{' followed by a (possibly empty) sequence of *variable declarations* followed by a nonempty sequence of *expressions* (with each expression followed by a semicolon) followed by a closing brace '}'.

A *main block* consists of the *main* keyword followed by a *variable-expression block*.

An *expression* can be any of, but **only**, the following:

- A plus expression (*expression1* + *expression2*).
- A minus expression (*expression1* - *expression2*).
- A times expression (*expression1* \* *expression2*).
- An equality test (*expression1* == *expression2*).
- A greater-than test (*expression1* > *expression2*).
- A not operator (!*expression1*).
- An or operator (*expression1* || *expression2*).
- A natural number (0, 1, 2, ...).
- The keyword *null*.
- An if-then-else expression having the form *if(expression1) {expression-list1} else {expression-list2}*, where *expression-list1* and *expression-list2* are nonempty sequences of expressions (with each expression followed by a semicolon).
- A for-loop expression having the form *for(expression1; expression2; expression3) {expression-list}*, where again, *expression-list* is a nonempty sequence of expressions (with each expression followed by a semicolon).

- A constructor expression having the form `new Classname()`. For example, `new Summer()` causes memory to be dynamically allocated and initialized for storing a `Summer` object.
- A `this`-object expression. As in Java, the keyword `this` in a method `m` refers to the object on which `m` was invoked.
- A print-natural-number expression: `printNat(expression1)`.
- A read-natural-number expression: `readNat()`.
- An identifier `id` (e.g., a variable name).
- A dotted identifier having the form `expression1.id`, where `id` is a field of whatever object `expression1` evaluates to.
- An undotted assignment having the form `id = expression1`.
- A dotted assignment of the form `expression1.id = expression2`.
- An undotted method call of the form `id(expression1)`.
- A dotted method call of the form `expression1.id(expression2)`.
- An expression inside a pair of parentheses: `(expression1)`.

Finally, comments may appear anywhere in a DJ program. A comment begins with two slashes (`//`). Anything to the right of the slashes on the same line is considered a comment and is ignored.

Again, you can find many example DJ programs illustrating this format at: <http://www.cse.usf.edu/~ligatti/compilers/22/a1/dj/>

#### 4 Key Differences between DJ and Java Programs:

- In DJ, semicolons must appear after every expression in expression sequences. Semicolons must even appear after `for` loops and `if-then-else` expressions. The example program above (in Section 2) illustrates this requirement with a semicolon after a `for` loop.
- In DJ, all field declarations in a class must appear before any method declaration; similarly, all variable declarations in a `variable-expression block` must appear before any expressions.
- The `main block` in a DJ program is not a method and cannot be invoked.
- DJ has no type for Booleans; we use natural numbers (i.e., `0`, `1`, `2`, ...) in place of Booleans (e.g., in `if-then-else` and `for-loop` expressions). The natural number `0` gets interpreted as `false`, and all other naturals get interpreted as `true`.
- DJ has no explicit `return` keyword. The example code in Section 2 illustrates how DJ uses the final expression in a method body to determine the return value.
- All DJ methods must take exactly one argument and return exactly one result.
- DJ classes have no constructor methods. DJ does have a built-in `new` expression, though: calling `new C()` creates a new object of type `C` having default values for all of its fields

(the default value for natural-number fields is *0*, and the default for object fields is *null*).

- DJ has no explicit *void* or array types and does not support type casting. The only types one can explicitly write in DJ are *nat* and object types.
- Natural numbers can be input and output using the built-in *readNat* and *printNat* functions.
- DJ requires all *if* expressions to have both *then* and *else* branches. For example, *if(0) {1;} else {2;}* is a valid DJ expression, but *if(0) {1;}* is not.
- DJ has no notion of *super*, *import*, *public*, *private*, *final*, *abstract*, *try*, *catch*, *throw*, *package*, *synchronized*, etc. It lacks all these keywords.
- DJ does not allow comments of the style */\* \*/*.

## 5 Additional Notes

### *Case sensitivity*

Keywords and identifiers are case sensitive (i.e., case matters, so "Class" is not the same as "class").

### *Identifiers*

Identifiers (which are used for naming classes, fields, methods, parameters, and local variables) must begin with a letter and must contain only digits (0-9) and ASCII upper- and lower-case English letters.

### *Natural-number literals*

All numbers in DJ programs have *nat* type and must be natural numbers (0, 1, 2, ...). Naturals may have leading zeroes in DJ; e.g., 005 is a valid *nat*, interpreted the same as 5.

### *The Object Class*

A class called *Object* is always assumed to exist. Class *Object* is unique in that it extends no other class. Also, class *Object* is empty; it contains no members (neither fields nor methods).

### *Recursion*

Methods and classes may be (mutually) recursive. A class C1 may define a variable field of type C2, while class C2 defines a variable field of type C1 (these are called *mutually recursive classes*).

### *Data Initialization*

All natural-number variables and fields get initialized to 0, and all object variables and fields to *null*.

### *Inheritance*

As in Java, classes inherit all fields and methods in superclasses. In DJ, subclasses may override *methods*, but not

*variable fields*, defined in superclasses. For example, if class C1 has a variable field *v1* and class C2 extends C1, then C2 may not declare any variable fields named *v1*.

A subclass may override a superclass's method only when the overriding and overridden methods have *identical* parameter and return types (though the overriding method's parameter name may differ from that of the overridden method). For example, if class C1 has a method *m* and class C2 extends C1, then C2 may declare a method *m* iff its parameter and return types match those of method *m* in class C1.

#### *How DJ programs evaluate*

DJ programs basically evaluate according to the rules for evaluating Java programs, with a few differences:

- *printNat* expressions evaluate to (and return) whatever natural number gets printed.
- *readNat* expressions evaluate to (and return) whatever natural number gets read.
- *for* loops, upon completion, always evaluate to (and return) the value 0.
- When the *then* branch of an *if-then-else* expression is taken, the entire *if-then-else* expression evaluates to whatever the *then* branch evaluates to. Similarly, when the *else* branch of an *if-then-else* expression is taken, the entire *if-then-else* expression evaluates to whatever the *else* branch evaluates to.
- Expression lists evaluate to whatever value the final expression in the list evaluates to.

#### *Dynamic (i.e., virtual) method calls*

As in Java, the exact code that gets executed during a method invocation depends on the run-time type of the calling object. For instance, the following DJ program outputs 2 because *testObj* has run-time type C2.

```
class C1 extends Object {
  nat callWhoami(nat unused) {this.whoami(0);}
  nat whoami(nat unused) {printNat(1);}
}
class C2 extends C1 {
  nat whoami(nat unused) {printNat(2);}
}
main {
  C1 testObj;
  testObj = new C2();
  testObj.callWhoami(0);
}
```

### *Assignment Expressions*

As in Java, DJ programs can make assignments to object-type variables. For example, the expression *obj1=obj2* causes the *obj1* variable to *alias* (i.e., point to the same object as) the *obj2* variable.

### *Typing Rules*

The typing rules for DJ also basically match those of Java. Beyond the normal Java restrictions, DJ requires that:

- The only types available to programmers are *nat* and object types.
- All class names must be unique.
- All method and field names within the same class must be unique. Although a subclass can override superclass methods, a subclass cannot override superclass variable fields.
- The *then* and *else* blocks in an *if-then-else* expression must have the same type.
- Boolean tests in the *if* part of an *if-then-else* expression, and in the conditional part of a *for-loop* expression, must have *nat* type (nonzero is used for *true* and zero is used for *false*). Similarly, *equality* (*==*), *or* (*||*), *greater-than* (*>*), and *not* (*!*) expressions all have *nat* (rather than *boolean*) type.
- A well-typed *for* loop has *nat* type (recall that it evaluates to 0 upon completion).
- *printNat* and *readNat* expressions have *nat* type because they evaluate to whatever number gets printed or read at run time.