

Meta-Ja

A Tool for Maintaining Large Java[tm] Projects
Version 2.16, of 2003-09-15

Martin Frank

<http://www.isi.edu/~frank/metaja>

USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292 USA

Java is a trademark of Sun Microsystems, Inc.

Meta-Ja is a trademark of the University of Southern California and of Martin Frank.

1 Introduction

Maintaining projects in Java by itself presents some progress over maintaining them in C++ in the sense that at least there is no duplication of function signatures between a "header" and a separate "implementation" file (there is still signature duplication for inherited functions). We have found that there used to be a large amount of "boilerplate" code that we tended to write (and - much worse - later had to maintain).

1.1 What does Meta-Ja help me with?

Specifically, we have found ourselves copying-and pasting code

- to make list functions type-safe - doing so means copying-and-pasting functions such as first() and then substituting the correct list item type as the return type because Java does not offer the equivalent of C++ templates
- to maintain an ordered list with simultaneous hash-based access for one or more attributes, such as a hash-based lookup by a "name" attribute - this can be achieved by overloading the list modification functions to maintain the hashtable(s), but doing so by hand is mechanical busywork and makes it easy to miss a function that should have been overloaded
- to create symbolic enumeration types - we have found that the best Java alternative to C++'s "enum" is to mechanically create a dedicated class which has an integer type as its lone data attribute and symbolic constants for the values of the enumeration
- to create the predictable getX() and setX() functions for variable name x - nearly all class attributes have accessors and many have modification functions
- to create the predictable constructors - many classes have constructors that take all its data attributes as arguments, and nearly all classes have constructors that take all those data attributes that do not have default values
- to create "carrier" classes for passing changeable arguments to functions - imagine you have an immutable type, say String, that you want to pass to a function for modification - since Java does not allow this, we mechanically create an encapsulating class called "StringCarrier" that has a String as its only data member, plus set() and get() functions on it - this class can then be passed and modified
- to maintain the list of included files and directories - all Java source files in a directory typically include the same list of include directories
- to create member functions required by Sun's JavaCC parser generator - it requires parsed classes to have seven jjtXxx() functions only one of which - jjtAddChild() - is actually relevant for non-list classes, and none of which is really needed for list classes

1.2 Why would I want to use it?

1. because you avoid authoring and maintaining boilerplate Java code as described above, obviously
2. because it eliminates the "dread of making up your own class hierarchy" - imagine that you want to quickly create a new representation with a few domain-specific functions on them, and also be able to parse the new representation from a file. Without the Meta-Ja Generator you may be tempted to e.g. choose an XML format for the stored representation, use an existing XML parser, and keep the representation in memory as untyped XML nodes - writing global functions that take XML nodes as arguments for the domain-specific functionality. As the amount of domain-specific code grows, you may quickly come to regret that seemed-convenient-at-the-time decision... with the Meta-Ja Generator, making up your own initial class hierarchy literally takes less than five minutes even for moderately complex ones (because you edit only a single "xxx.mjPackage" file, specifying only the class hierarchy and class data members).

1.3 Why would I *not* want to use it?

1. because you have wedded yourself to a proprietary Java development environment such as Symantec Visual Cafe or Borland JBuilder which may not have a "make"-like facility - you would have to manually invoke the Meta-Ja generator to first generate the .java files before you then generate .class files from them - that could be painful (if you are on Unix you already have "make", and you can download one for the PC from <http://sourceware.cygnus.com/cygwin>, but then you probably still have to use "make" from a command-line - as opposed to from a button within your development environment (in Emacs, M-x compile is already bound to invoke "make" by default, of course)
2. because compilation error messages and run-time exceptions refer to line numbers in the generated .java files rather than the .mjCode files - which means that automated error look-up functions as in Emacs bring up the wrong file to edit
3. because you don't want to make yourself dependent on a volunteer-maintained tool - you have two hedges against that problem. First, you can simply throw away the tool and continue by hand-editing the generated Java files. Second, you can license the Meta-Ja source code so you can extend/maintain/debug your own private version from some point in time if you so choose.

2 Installation Instructions

2.1 The Distinction between the Utilities and the Generator

Meta-Ja consists of two distinct software modules - the *Meta-Ja Utilities* and the *Meta-Ja Generator*.

The *Meta-Ja Utilities* are a run-time library (currently 429073 C: bytes) that is linked in with the generated Java files to produce your final product. It contains basic programming abstractions beyond the ones that come with Sun Microsystems' Java Development Kit (JDK).

The *Meta-Ja Generator* is needed at design time to produce Java source code files from your higher-level Meta-Ja files. It does not need to be distributed to your customers in any form.

Below is a summary of these two components.

	Utilities	Generator
needed at run-time	yes	no
needed at design-time	yes	yes
click-through-license	object	object
commercial standard license	source	object
commercial source license	source	source
re-distribution		
with final product	object	no

2.2 The Contents of your *mjdistribution.jar* file

Your *mjdistribution.jar* file contains the following files.

1. *mjutilities.jar* **Do not un-jar further!** Contains the binary files for the Meta-Ja Utilities.
2. *mjgenerator.jar* **Do not un-jar further!** Contains the binary files for the Meta-Ja Generator.
3. *metaja.pdf* A copy of the very Meta-Ja documentation that you are currently reading (but more likely to be in sync with this distribution's binaries).
4. *utiljavadoc.jar* Contains the JavaDoc-generated documentation of the Utilities classes and methods. (You can also browse them at our Web site, but this version is more likely to be in sync with this distribution's binaries).
5. *mjexamples.jar* Some example .mjPackage and .mjCode files.

2.3 Getting Started

Create a new directory called "test".

```
mkdir test
cd test
```

Copy the mjddistribution.jar there.

```
cp wherever/you/downloaded/it/to/mjddistribution.jar .
```

Now un-jar the distribution file (assuming you put Sun Microsoft's JDK *bin* directory in your PATH, otherwise you may have to invoke *jar* with something like `$JDK_HOME/bin/jar`).

```
jar xvf mjddistribution.jar
```

Un-jar the examples file:

```
jar xvf mjexamples.jar
```

Un-jar the JavaDoc documentation. (You can then read it by pointing your Web browser to the file "utiljavadoc/index.html".)

```
jar xvf utiljavadoc.jar
```

Create your own example project directory structure.

```
mkdir com
cd com
mkdir me
cd me
mkdir zoo
cd zoo
cp ../../../../edu/isi/dce/mjexamples/basezoo/* .
```

Make Meta-Ja create the Java files (note that the classpath syntax below is for Windows, using a semicolon to separate entries; most other operating systems use a colon instead).

```
java -classpath ../../../../mjutilities.jar\;../../../../mjgenerator.jar \
  edu.isi.dce.mj.MjRep -classpath ../../../../mjutilities.jar \
  -currentDir $PWD -topDir com
```

Note that the first classpath argument is for running Meta-Ja itself, while the second one is the one you will compile the generated Java files with (because Meta-Ja looks into the directories, zip, and jar files of the classpath for `.mjPackage` files). The `-currentDir` and `-topDir` arguments are needed for Meta-Ja to figure out the appropriate package name for the directory ("com.me.zoo" if you followed my instructions to the letter).

Compile the Java files.

```
javac -classpath ../../../../mjutilities.jar *.java
```

Run the example. (The first entry in the classpath is for the supporting Meta-Ja utilities library, and the second so that the `ZaZoo main()` function can be found.)

```
java -classpath ../../../../mjutilities.jar\;../../../../ com.me.zoo.ZaZoo \
  -text whatever -truth false -number 123 -oneAge young
```

Yes, I know - you got an error because class "penguin" cannot be read back in from XML, and only an elephant and a giraffe appear in the XML output. This is because we did not compile a Java class for it - so there is no need to send me e-mail about it. (If you are bothered by it, either (1) remove the penguin data from the `exampleZoo.xml` file, or (2) create a sister directory "derivedzoo" parallel to "zoo", copy the `.mjPackage` file from `edu/isi/dce/mjexamples/derivedzoo` there, compile it, and modify the "f" tag in the `exampleZoo.xml` file to point to it.)

This should get you started on your project. You can clean up the Meta-Ja-generated files by adding "clean" to the command line above that produced them.

3 Tutorial

Meta-Ja relies on one text file per directory/package for describing the classes that are part of that package. This file always carries the *.mjPackage* extension ("Meta-Ja package description file").

For the sake of this example, suppose you are writing Java classes for a custom database for keeping track of animals in a zoo. We will name the package file "Za.mjPackage" because we want to prefix all class names in this packages with "Za" (Meta-Ja will deduce that prefix from the file name of the package description file but you can also explicitly specify a different prefix, or use no prefix at all).

Here are the contents of this file.

```
abstract object Animal {String name;}
object Elephant extends Animal {}
object Giraffe extends Animal {int lengthOfNeck;}
list<Elephant> Elephants {}
list<Animal> Animals -key name {}
```

You invoke the Meta-Ja generator by going to the directory in which the *.mjPackage* is located and saying "java edu.isi.mastermind.Utilities.MmData.MmDataRep" (for any real work, you will obviously automate that invocation via a Makefile). In our example, the Meta-Ja generator will write out the files *ZaAnimal.java*, *ZaElephant.java*, *ZaGiraffe.java*, *ZaElephants.java*, and *ZaAnimals.java* (as well as a *.cvsignore* file that tells the CVS version control system that these are generated files). All of these files include three standard directories: *java.util*, *java.io*, and *edu.isi.mastermind.Utilities* (we will later see how to suppress the inclusion of the standard directories, and how to specify additional ones).

Each of the non-list classes contains the specified data members (e.g. "private String iName;"), accessor functions for these data members (e.g. "public String getName() {return iName;}"), constructors for the classes (e.g. "public Giraffe(String name,int lengthOfNeck) {...}"), copy functions "public ZaElephant copyZaElephant() {...}", and print functions "public void print(Writer w) {...}").

Each of the list classes is subclassed from the abstract *edu.isi.mastermind.Utilities.BasicList* (which is really just a JDK *LinkedList* with additional functions), but provides type-safe versions of all functions (in much the same way as C++ templates do). In addition, the *ZaAnimals* list also implements hash-based look-up on the name attribute as specified, and overloads list modification functions to keep the name hashtable up to date.

You can specify a package directive and additional include files by placing the compilation directives below at the very beginning of the *Z.mjPackage* file. The prefix declaration below is redundant in this particular example but shows you how to explicitly specify a prefix for all generated classes (prefix "" - using no prefix at all - is also a valid directive).

```
compilationDirectives {
    package "edu.isi.ZooAnimals"
    import "edu.isi.mastermind.AdaptiveForms.Grammar"
    import "edu.isi.mastermind.AdaptiveForms.ParseTree"
    prefix "Za"
}
```

The generated code will fully implement the class hierarchy you described, and you can compile it as is, but most likely you made up the class hierarchy for a *reason* - you want to attach domain-specific functions to the classes. This is done by putting your additional code in e.g. `ZaAnimal.mjCode` ("**Meta-Java Custom Code**"). You do not have to specify the enclosing `"class ZaAnimal {"` and `"}"` - The generator will simply copy the contents of this file verbatim into the class body after the automatically-generated members. For example, the following custom print function could be added to class `ZaAnimal` via a file **`ZaAnimal.mjCode`**:

```
public void myPrint(Writer oo) {
    PrintWriter o = new PrintWriter(oo);
    o.print("I'm elephant \""+iName+"\"");
    o.flush();
}
```

In the simplest case, this is all there is to it. You can remove the generated code via `"java edu.isi.mastermind.Utilities.MmData.MmDataRep clean"` (you want to automate this via a Makefile in real life as well).

We strongly suggest at this point that you download and install the Meta-Ja generator and experiment with your own class hierarchy and examine the generated code. There is no need for you to read this entire manual before you start using Meta-Ja.

Hand-written Java files and Meta-Ja generated co-exist peacefully even within a single directory, so you can for example just make a single list Meta-Ja-generated while keeping all your existing code as is, and migrate to Meta-Ja use at a pace that you are comfortable with.

4 Reference Manual

This is a complete list of all features implemented, vaguely in the order in which we believe you will come to use them as you become more comfortable with Meta-Ja.

4.1 Influencing the naming of types

If you refer to a type T in a `.mjPackage` file whose prefix is Xx , the Generator will translate type T into " XxT " in the generated Java file, unless T matches a list of predefined types such as "String" and "boolean" in which case it will be left as is.

You can force the Generator to leave the type name as is by prefixing it with an exclamation mark, as done below:

```
object SubmitFormAction extends !EgAgSubmitFormAction {}
```

You can also force the Generator to always prefix a type name - even if it matches a predefined type name such as "String" - by prefixing it with a dot. (You could also achieve the same effect by prefixing the type with an exclamation mark and the prefix for this directory, but the dot notation is somewhat more elegant.) The only reason to do so is because you are defining a type in this directory that (without the directory prefix) matches the name of a predefined type (we have so far never encountered such a case).

```
object String {}
object Referer {String a; !String b; .String c; !XxString d;}
```

In this example, the generated type names will be "String" (because it matches a name that the Meta-Ja generator knows is predefined), "String" (the user has explicitly asked for that type name to be left alone), "XxString" (the user explicitly wanted us to prefix the type) , and "XxString" (the user has explicitly asked for that type name to be left alone), respectively.

4.2 Omitting list names and data member names

You can omit the name of lists in `.mjPackage` files and a (normally) sensible one will be generated for you.

```
object Giraffe {String name; int age;}
list<Giraffe> {}
```

The list of Giraffe above had its name omitted, which is equivalent to having said "`list<Giraffe> Giraffes {}`". (The Meta-Ja generator can nearly always deduce the plural of a list item name: animal->animals, datum->data, status->stati, entry->entries, class->classes.)

You can omit the name of data attributes if the attribute type is unique within the class. In this case, the name of the type will also be used as the name of the attribute.

```
object Inventory {
  Apes;
}
```

This is entirely equivalent to having said:

```
object Inventory {
  Apes apes;
}
```

4.3 Default values for data members

Data attributes will be initialized with *null* for all object types (including `String`), *false* for booleans, *0* for numeric values, and ' ' (space) for characters and bytes. However, you can also provide an expression for initialization in double quotes after the variable name, which will be copied verbatim into the generated Java code.

```
object Inventory {
  String name "\"default name\"";
  String category "someStaticStringConstant";
  Elephants herd1 "new ZooElephants()";
  Elephants herd2 "";
  Giraffes myGiraffes "Inventory.createSomeDefaultGiraffes()";
  AuditType "full";
}
enumeration AuditType {"selective","full"}
```

In general, whatever you provide as the initialization `String s` will be copied verbatim into the class initialization of the form `"Type name = s;"`.

A common mistake for *String-valued attributes* is to not quote constant text twice, as shown for `name` above.

For *list attributes*, initializing a list to `"` (the empty string) is equivalent to explicitly making it the empty list (`"new ZooElephants()"` in the above example).

For *enumerations*, initializing an enumeration `RestaurantType` to `"full"` is a shorthand for `"AuditType.createFull()"`.

Note that changing which data members have defaults changes the function signatures of the generated constructors, as explained in the next section.

4.4 Influencing the generation of accessors, modifiers, and constructors

The Meta-Ja Generator will produce default constructors for each class unless you ask it not to. There will be up to four constructors generated:

1. one taking all inherited and all local attributes as arguments
2. one taking non-default inherited and non-default local attributes
3. one taking all inherited and non-default local attributes
4. one taking non-default inherited and all local attributes

If any of these four potential constructors have identical arguments, only one of them will be generated. These heuristics prevent writing out a number of constructors exponential in the number of attributes - and in our experience virtually always (>95%) provide what you need. (You can always add your own constructors by hand, of course.)

You will sometimes want to use "unnecessary" initializations in your `.mjPackage` file just to influence the generated constructors. For example, both `String` attributes below will default to `null` - explicitly declaring the `careTaker` to be `null` lets the generator know that it's ok to construct an `Animal` object without providing a caretaker (thus it writes the constructor `"public Animal(String name) {iName=name;}"` which it would not have written without the `null` declaration of `careTaker`).

```

object Animal {
    String name;
    String careTaker "null";
}

```

You can suppress the generation of all constructors with the `"-noConstructors"` directive. You may want to do this if you do processing in constructors (which I would generally discourage, especially if the processing can result in a failure condition), or because you want to acquire external resources (such as initializing access to a remote data base).

```

object MyAction extends Action -noConstructors {}

```

You can similarly suppress the generation of the `getX()` function for data member `x` (`"-noGet"`), and request the generation of a `setX()` function (`"-set"`). The reason for this asymmetric treatment is that by far the most common case is that there is a `get()` but no `set()` function for data members.

```

object Whatever {
    boolean selected -isGet -set;
    boolean highlighted -bareGet;
    boolean processed -noGet;
    boolean active;
    Inquiry -set;
}

```

This results in the access functions `isSelected()`, `highlighted()`, `getActive()`, and `getInquiry()`, and in the modification functions `setSelected(boolean)` and `setInquiry(Inquiry)`.

4.5 Enumeration Classes

C++ offers the `"enum"` construct for symbolic enumerations but Java has no equivalent (at least so far). We have found that the best solution is to create a custom class which privately holds a single integer data member plus constants for each of the cases of the enumeration. We then generate test member functions such as `"boolean isLocal()"` and modification functions such as `"void setToLocal()"`, as well as static creation functions `"static XxChange createLocal()"`. Below is an example enumeration declaration. Note that you can add custom code to generated enumeration classes via e.g. a `"XxChange.mjCode"` file as usual.

```

enumeration Change {"none","local","structural"}

```

4.6 Carrier Classes

We have found that we sometimes want to pass a `"pointer-to-a-class"` to a function that the function can modify to point to a different class. You can have the Meta-Ja Generator create a `"carrier"` class for that purpose. A carrier class contains a reference to the original class as its only data member, together with `get()` and `set()` functions to modify it. Below, we request the creation of an `"InterpreterCarrier"` class in addition to the regular `"Interpreter"` class.

```

object Interpreter -carrier {}

```

4.7 Making Generated Constructors and Functions Non-Public

By default, Meta-Ja's generated constructors are public. You can make them private (for access by this class only), "restricted" (for access in this package only), or protected (for access by this package or by subclasses).

```
object X -privateConstructors {}
object Y -restrictedConstructors {}
object Z -protectedConstructors {}
```

It is generally desirable to invoke Meta-Ja-generated constructors from your custom-written constructors. This is because they break if you e.g. add a member variable in the .mjPackage but forget to update your hand-written constructor(s) to initialize it, making this a compile-time issue rather than a debug-at-run-time issue.

For that reason, there is a special flag for generating private constructors so that they do not conflict with constructor signatures you want to write by hand.

```
object X -augmentedPrivateConstructors -noGenFct {int a;}
```

In this example, you want to define an X(int) constructor by hand, say to do some transformation on the input value. (You will have to say -noGenFct above if you do not want to supply the constructors that Meta-Ja normally expects by hand. Alternatively, you can supply the regular constructors without the byte argument by hand, and then the Meta-Ja generated XML reading and copying functions will work as usual, invoking your hand-written constructors as they go along.) The -augmentedPrivateConstructors flag will generate the usual constructors but append an un-used byte-typed argument to them:

```
private X(int a,byte argumentsAugment) {iA=a;}
```

Thus you can now again use a Meta-Ja-generated constructor for your hand-written constructor, as follows.

```
public X(int a) {this(transform(a),(byte)0);}
private static int transform(int a) {/*...*/}
```

4.8 Making Generated Get and Set Functions Non-Public

By default, Meta-Ja's generated get() and set() functions are public. You can make them non-public with the per-data-member "-restrictedGet", "-protectedGet", "-restrictedSet" and "-protectedSet" directives. Here is an example:

```
object X {
  int a -set -protectedGet -restrictedSet;
}
```

4.9 Automatic Command-Line Parsing

Meta-Ja can generate code for parsing command-line arguments for you. The key advantage of Meta-Ja doing that (as opposed to doing the same thing with a simple library routine) is that Meta-Ja gives you the argument values *already appropriately typed* - there is no need to cast from String-valued arguments to boolean, Integer, or whatever you were expecting. This is analogous to its advantage of giving you strongly-typed Java classes when parsing XML, as opposed to you having to author a translation step from String-based XML parser output to your native classes (using the SAX API or similar).

You do this by adding the `-commandLine` flag to any `.mjPackage` object definition that contains attributes of built-in or enumerated types. The object can inherit such attributes from another object, which is handy if you have several programs that take some common command-line arguments (note that in that case, you don't have to say `"-commandLine"` on the abstract classes – only on the leaf classes). Here is a simple example, say in a file called `"Xx.mjPackage"`.

```
enumeration Age -xml {"young","midCareer","old"}
object Args -commandLine
{
    String text;
    int number;
    Age oneAge;
    Age anotherAgeWithADefault -set "midCareer";
    boolean truth;
}
```

This causes the Meta-Ja generator to produce the following code in `XxArgs.java`:

```
public static XxArgs parseCommandLine(String[] argsRaw) {
    return parseCommandLine(new Strings(argsRaw));
}
public static XxArgs parseCommandLine(Strings args) {
    return parseCommandLine(args,true,true,true);
}
public static XxArgs parseCommandLine(
    Strings args,
    boolean echo,
    boolean processSpecialSwitches,
    boolean last/*command line options*/)
{
    /*...*/
}
```

The last function takes the full set of arguments for the command-line parsing. *args* is the raw list of strings from the command line. If *echo* is set to true (the default), the value of the command-line arguments will be echoed to standard error. If *processSpecialSwitches* is set to true (the default), the function will look for the special switches `-version` and `-help`, and print out a message and terminate execution if one of them is found. Finally, if *last* is true (the default), the function will warn about any flags that it could not interpret and terminate execution.

The reason for the *last* flag is that you can chain several calls to command-line parsing functions. If you call one of the variations that takes a `Strings` argument (not `String[]`), each such call will remove the flags that it knows about. The last and only the last invocation should have the *last* argument set to true if you want to check for mistyped command line arguments.

Here is a simple invocation for the above example.

```
public static void main(String args[]) {
    XxArgs x=XxArgs.parseCommandLine(args);
    if(x.getTruth()) {Util.out.println(x.getNumber());}
}
```

And here is a more complex, chained invocation.

```
public static void main(String rawArgs[]) {
    Strings a=new Strings(rawArgs);
    XxArgs x=XxArgs.parseCommandLine(a,true,true,false);
    YyArgs y=YyArgs.parseCommandLine(a,true,true,true);
}
```

Here is the output for the special `-version` and `-help` flags.

```
$ java edu.isi.mjexample.XxMain --version
Compiled 2001-02-05T12:02:31-08:00 on bugatti/128.9.128.183 by frank
with Java Development Kit 1.3.0 under Windows 2000.
$ java edu.isi.mjexample.XxMain --help
XxArgs:
Required:
  -text null -number 0 -oneAge null -truth false
Optional:
  -anotherAgeWithADefault midCareer
```

Note that `-help` will only print the arguments of the first invocation of a chained invocation of `parseCommandLine()` functions (because your `main()` function can decide which subsequent invocations happen based on the value of prior arguments, for example!). Thus, if you chain parsing functions at all you should implement your own command-line help function.

Note also that Meta-Ja could in principle provide more helpful help, by listing the possible enumerated values of enumeration-typed arguments for example. It would also be nice if the user could provide a "good" value for required arguments in the `mjPackage` file so that Meta-Ja does not have to use just the type default (0, false, 0L, null, ...) which is not particularly helpful.

4.10 Auto-Implementation of hash-based access to list items

We have found ourselves implementing hash-based access to items in a list based on a unique data member over and over, and the same is true for standard-issue `"find()"` and `"contains()"` functions on lists. Therefore, the Meta-Ja Generator can now automatically implement the above.

Just as in the database world, a data attribute can uniquely identify an item (e.g. `"-key[unique] id"`) or not (e.g. `"-key[multipleMatchesPossible] content"`). If the attribute is unique, the generator will generate an `"ItemType findId(String x)"` function, otherwise a `"ListOffItemtype findContent(String x)"` function (as there may be more than one matching items if the attribute in the latter case). The default is "unique".

You can declare up to one key in a list class as the primary key (`"-key[primary] id"`). The only effect is that this will generate a `find()` function that is identical to `findId()`, the point being that if the find criterium is clear there is no need to have the data member's name in the function. The default is "primary". (So if you have two unique keys for a list, you have to declare at least one of them secondary as in `"-key id -key[secondary] name"`, or there will be a duplicate generated `"find()"` functions!)

For completeness of discussion, the implementation of keys can be directed to be hash-based `"-key[hashed] id"` or not `"-key[notHashed] id"`. There is rarely a reason not to use

hashing in practice but it is possible in principle that you may want to do this to trade storage space for access speed.

The default is "hashed" (for keys that are actually unique only, of course).

```
object Pair {String name;String value;}
list<Pair> Pairs
  -key[unique,primary,hashed] name
  -key[multipleMatchesPossible,secondary] value
{}
```

In the above example, the generated Pairs list class will maintain a hashtable over the Pair name whenever the list is modified and offer fast "Pair findName(String x)" and "Pair find(String x)" functions. It will also offer a non-hashed "Pairs findValue(String x)" function.

In the case that the key attribute is inherited from outside the current directory, the Generator cannot know the type of the attribute. In that case, you can explicitly tell it about it through a "type=XXX" flag as shown below. In that case, the Generator will take your word for it - if you misspell the type the generated code will not compile.

```
object Animal extends !NamedObjectDefinedOutsideThisDirectory {}
list<Animal> Animals -key [secondary,type=String] name {}
```

4.11 Typed List Iterators

For every list class such as *Animals* there is also a generated *AnimalsIterator*. This iterator implements the JDK *ListIterator* interface, but its functions are type-safe so that you e.g. do not have to cast from *Object* to *Animal* when you call *nextE()*. In addition, the generated list iterators ensure that any automatically maintained hash tables are maintained if you call *removeE()* on the iterator.

4.12 Working with Sun JavaCC's .jdt parser generator files

A popular tool for implementing parsers is Sun's JavaCC ("http://www.sun.com/suntest"). JavaCC requires that classes that are parsed into have a public constructor that takes a single integer as an argument. In addition, they are seven mechanically-implementable functions that it requires (*jdtOpen()*, *jdtClose()*, *jdtParent()*, *jdtSetParent()*, *jdtGetParent()*, *jdtGetChild()*, *jdtGetNumChildren()*). All of the above will be generated for you if you declare a class to be "-parsed" in the *.mjtPackage* file, as shown below.

```
abstract object PrefixFiller {}
object PrefixLiteral extends PrefixFiller -parsed {
  String value;}
object PrefixRegex extends PrefixFiller -parsed {
  String value;String category;}
object PrefixNonTerminal extends PrefixFiller -parsed {
  String name;PrefixFillers fillers;}
list<PrefixFiller> -parsed {}
```

For non-list classes, you will have to add JavaCC's required *jdtAddChild()* function. For list classes, the generator creates a *jdtAddChild()* function that implements pushing the new element in the list.

4.13 Interfaces

You can define Java interfaces via the *interface* keyword, as shown below, much as you define regular Java classes (identified with the "object" keyword in Meta-Ja). Custom code (which in the case of interfaces by definition can only be abstract functions) can be added in a `MyActionListener.mjCode` file as usual.

```
interface MyActionListener {}
```

4.14 Declaring abstract functions

You can declare abstract functions in the `.mjPackage` description of object or interface description *X* if you wish. You don't have to - you can also put them into the corresponding `X.mjCode` file instead. We typically prefer to put them into the `.mjPackage` file, especially for interfaces because that communicates why they exist. The rationale for putting them into the `.mjCode` file is to not clutter the `.mjPackage` file, for the reason that which we also find `.mjPackage` files also serve as a convenient overview of what a particular package does.

Below are some example function definitions. Note that the "restricted" keyword that is equivalent to omitting any access modified in a plain Java file (not specifying either `public`, `protected`, or `private`).

There is no need to declare any of these functions "abstract" - the Meta-Ja generator knows that all functions declared in a `.mjPackage` file are abstract by nature.

```
object ZooAnimal {
  restricted void feed(int howMuch);
}
interface RevenueSource {
  public !Currency distributeCash();
}
```

You can omit argument names in function declarations, but you should only do this if there is only one variable of that type in the function, and adding a name would not give the reader any more information. In the example below, providing symbolic names for the arguments of the first section is crucial, but providing it for the second function would be silly ("`ListElements listElements`" doesn't provide any additional information).

```
object Representation {
  public void set(String name,String value);
  public void processListElements(ListElements);
}
```

4.15 Comments in `.mjPackage` files

You can embed comments in `.mjPackage` files. These comments are especially designed to translate well into comments that Sun Microsystem's JavaDoc uses to generate Java code documentation.

There are several stages of comments in a `.mjPackage` file (in `.mjCode` files, follow the standard JavaDoc comment conventions). At the very beginning of the `.mjPackage` file (even before any *compilationDirectives* declaration) there are comments applicable to the entire package, and there are class-level comments which go immediately before the "{"

of *objects*, *interfaces*, or *enumerations*. Comments on variables go just before the ";" of variable definitions. For abstract functions, there are comments for the entire function, which also go just before the ";", and there are comments about the return type declaration, which go just after the return type, and finally comments on function arguments, which go just after their declaration.

Below is an example of all these comments. The Meta-Ja generator knows how to translate these comments into the format that JavaDoc expects.

```

/* implements a data base for zoo animals */
compilationDirectives {
    prefix "Za"
}
abstract object Animal /*base class for all life forms*/ {
    String name /*unique identifier for an animal*/;
}
interface RevenueSource /*implemented by all cash generators*/ {
    public !Currency/*in US dollars*/ distributeCash(
        String originator /*who sent it*/,
        String clearingHouse /*the bank it went through*/)
    /*transfers cash into the Zoo's data base*/;
}
enumeration SourceStatus
/* explains where the value of a slot came from */
{ "untouched" /*no value for this slot anyway*/,
  "beingComputed" /* currently being computed by system */,
  "computedBySystem" /* was computed in the past */
}

```

In addition, you can place C++-style "//"-comments anywhere in a Meta-Ja package file; these are not passed on in any way to the generated code.

4.16 Generating list classes from a sub-classed BasicList

The generated list classes normally inherit from *BasicList* in the edu.isi.mastermind.Utilities package. You can specify a different list class to inherit from in a second angular bracket as shown below. The only situation where you may ever want to do this is if you want to derive from BasicList and overload the modification functions of the list by hand, and then have the Generator create type-specific list classes that inherit from that modified list. We have used this feature only once, for tracking list modifications. Below is an example:

```
list<MainTableElement><<!ModBasicList> MainTableElements {}
```

4.17 Automatic Package Names

If you add "-currentDir //C/Code/edu/isi/ZooAnimals" and "-topDir edu" to the invocation of the Meta-Ja generator, it will infer that the package name should be "edu.isi.ZooAnimals" (you can override this package name by explicitly specifying the package name as usual in the compilation directives, and you can still produce non-package Java code by setting the package name to the empty string). This is

particularly handy if you have a large number of sub-directories in your project and you are still moving directories around. In real life, you would obviously use a project-level Makefile included by the directory-specific Makefiles that automatically provides these command-line flags rather than typing them by hand.

4.18 The Auto-Generated Copy, Print

There are certain types of functions that are nearly always desirable for your classes and which can be mechanically implemented. In Meta-Ja, these include functions for making a copy of a class and for printing out classes in a standard fashion.

Meta-Ja will generate these functions for all regular Java classes (the ones prefixed with "object" in their package description file), unless you explicitly ask it not to by specifying the "-noCopy" or "-noPrint" flags in its object description (or just "-noGenFct" to suppress all of them). The only reason we can imagine is that you may not want to generate them is because (1) you are concerned about the memory or disk space consumption of functions that may not be needed anyway or (2) a generated function name clashes with the function name of a non-Meta-Ja-generated class that you are inheriting from.

In some cases, you may also want to implement a normally generated function yourself, for example if the mechanically-produced print() version always prints out a large hashtable that you really don't care about. In this case, you can use the "-customPrint" flag and produce your own implementation, most easily by just copying the mechanically generated functions into the .mjCode file and then modifying it. In case you are wondering why "-customPrint" is any different from "-noPrint" the answer is that when class B later lists this class A as a member variable, Meta-Ja can only print the memory location of A in B's auto-generated print function, but will call your custom implementation in the former case. There are analogous "-customCopy", "-customPrint", and "-customGenFct" flags.

The generated Java interfaces (the ones prefixed with "interface" in the package description file) will not contain signatures for these functions unless you explicitly ask for them via "-genFct" (or a la carte via "-copy" and "-print"). This is because you will normally not want to burden external implementors of your interfaces with having to implement these extra functions. You may want the abstract function headers to be generated into the Java interface code if (1) you use the interfaces just within your own code, if (2) you know that your users also use Meta-Ja and you make your .mjPackage files available to them, or if (3) you rely on one of those functions in your code and thus want your users to implement them, even if they have to do so by hand. The additional advantage of having these functions available for interfaces also is that Meta-Ja can then e.g. again call its print() function from the generated print() function of an embedding class.

4.18.1 Copy Functions

Each Meta-Ja generated class X has a function of the form "public X copyX(int depth) {...}" which implements a memberwise copy functionality. The function cannot be simply called copy rather than copyX because it is desirable to have copy functions with different return types in an inheritance chain yet Java does not allow overloading a function name by return type (it does only for argument types). For example, assume an inheritance hierarchy where class C is derived from abstract class B which is in turn derived from abstract class

A. In this case, it is desirable to be able to call `copy` on A (with return type A) as well as on B (with return type B). In either case, Meta-Ja will implement all three copy functions automatically (in C, they all have the same functionality, just different return types).

The depth of the copying is indicated with an integer parameter. For example, consider the following package definition.

```
object NVPair {String name;String value;}
list<NVPair> NVPairs {}
list<NVPairs> NVPairsList {}
```

Presume you have an instance *l* of type `NVPairsList`. What does it mean to "copy" it?

- You could just copy the pointer *l* itself to *c*.
- You could make *c* the empty list and then copy the top-level pointers of *l* into it.
- You could make *c* the empty list, then instantiate a new `NVPairs` copy for each `NVPairs` pointer in the original `NVPairsList`.
- Finally, you could make copies of all the top-level `NVPairsList`, of all `NVPairs` and of every `NVPair` itself. In that case you made a complete deep copy, and could throw away the original *l* without harming your new copy in any way.

These types of copies are of depth 0, 1, 2, and 3, respectively, as far as the Meta-Ja copy functions are concerned. In addition, telling a Meta-Ja copy function to make a copy of depth -1 is equivalent of making a copy of infinite depth.

4.18.2 Print Functions

Finally, every regular Java class will auto-implement "*public void print(Writer w,String indString,int indLevel)*". Its primary purpose is in having an easy way to print out the content of your classes when you are debugging your code. For example, a `ZaElephant` instance may output the following for a `print(Util.err,".+:",2)` call:

```
.+:.+:ZaElephant {
.+:.+:.+:name "dumbo"
.+:.+:.+:pastIllnessDates {
.+:.+:.+:.+: "8/26/98"
.+:.+:.+:.+: "11/4/98"
.+:.+:.+:.+: "2/12/99"
.+:.+:.+:}
.+:.+:}
```

Typically, the indentation string will of course be a couple of space characters and the initial indentation level will be zero. For that reason, there will also be an auto-implemented "`public void print(Writer)`" function which is equivalent to calling `print(w," ",0)`.

4.19 XML Reading/Writing Functions

You can make any Meta-Ja defined entity (object, interface, list, or enumeration) `Xml`-capable by simply adding the `-xml` flag to its `.mjPackage` declaration. In that case, the entity `E` will have a "`static public E readFromXml(Reader r)`" method and a "`public void writeXml(Writer w)`" method.

An individual data member of an object can be declared *-volatile*, which means it will not be written out or read in from the XML representation (for example, if you keep some list around that is only used at run-time for internal processing purposes). Note that if declared volatile, the member will not be copied or printed in the other Meta-Ja-generated functions either.

To be XML capable, your Meta-Ja objects must contain a constructor that takes all the non-defaulted, non-volatile arguments of its inherited and local data members, in order. Meta-Ja writes such a constructor for you anyway, but if you explicitly disable it with *-noConstructors* you become responsible for writing one by hand because XML reading depends on it. In addition, each optional (non-defaulted) data member that you want to write out and read back in must sport a *set* function known to Meta-Ja (either a set function produced by Meta-Ja via *-set* or *-voidSet*, or a set function that you wrote and made known via e.g. *-customSet* "setCareTaker") - if it does not Meta-Ja will treat it as if it were declared *-volatile* (and put a comment about that in the generated .java file).

The functions will not be written out if there is a data member in the entity that was defined in a .mjPackage file, was not declared *-volatile*, and that had its own Xml functions suppressed (or not enabled in the first place in case it was an interface). Note that a copy function will not be written out for that case either. A comment about why the function was not written out and how to enable it will be generated into the .java file.

The generated XML reading function will return *null* if any of the required data members (the non-defaulted ones) of the object are missing in the XML text stream. It will also return *null* if any of its optional data members were present in the parsed XML but could not be reconstructed. This sledgehammer approach is much superior to silently returning corrupted content; but it would be nice to have more control over what to do if a parsing exception occurs in the future (stop,continue with a warning,continue silently, etc...).

4.20 Reference: all compilation directives

The following directives can be issued at the beginning of a file, between *compilationDirectives*{ and }.

4.20.1 *import*

Fulfills the same functions as *import* in plain Java, with the exception that it denotes entire packages – there is an implicit *.** behind any Meta-Ja import directive.

```
compilationDirectives {
    import "edu.isi.dce.mjexamples.basezoo";
}
```

Thus, the above is equivalent to importing "edu.isi.dce.mjexamples.basezoo.*" in all plain Java files of that directory.

4.20.2 *xmlimport*

Your multi-package Java code typically benefits from having a clearly defined *compile order*; that is, a linear order in which your packages can be compiled one by one, without circular references.

However, there is one case where you may want to have Meta-Ja "look ahead" to find out which subclasses are known for an abstract class or interface. This is when you want to be able to read a data structure from XML, and be able to read these subclasses without having to give them "f" tags in the XML (see also the section on XML).

In these cases, you can use the following statement.

```
compilationDirectives {
  xmlimport "edu.isi.dce.mjexamples.derivedzoo";
}
```

This makes sense if the listed directory contains a concrete "-xml"-declaring subclass of a local "-xml"-declaring abstract class or interface (but Meta-Ja does not check for that, as otherwise the statement does no harm). This way, you can look ahead without using the *import* statement which would have the very undesirable side effect of introducing a circular import dependency.

4.20.3 *customJavaFile*

It is perfectly legal to mix Meta-Ja generated Java files and hand-written Java files in the same directory. If you do, let Meta-Ja know about it via the *customJavaFile* directive. Note that the ".java" postfix is implicit.

```
compilationDirectives {
  customJavaFile "MyFile";
}
```

This way, Meta-Ja will not issue a warning because it believes that the .java file is a result of you renaming a class in a .mjPackage file.

4.20.4 *ignore*

Meta-Ja generates a .cvsignore file for you so that for example, its generated Java files are ignored as far as the Concurrent Versions System (CVS) is concerned, but not hand-written Java files in the very same directory. You can ask Meta-Ja to put additional lines into the .cvsignore file verbatim, as shown below.

```
compilationDirectives {
  ignore "one.log two.log three.log";
  ignore "*.tmp";
}
```

There is no other effect of an *ignore* directive. Note that you should typically not use *ignore* for Java files, use *customJavaFile* instead.

4.20.5 *prefix*

Rarely used. Meta-Ja usually uses the name of your .mjPackage file as the prefix for classes in that directory. Thus, if your .mjPackage file is named "Za.mjPackage" and defines an object called "Animal", the generated class name will be called "ZaAnimal".

You can change the prefix with an explicit directive below, but it is rarely a good idea to do so because short, unique prefixes help prevent confusion because you don't have e.g. a "Params.java" file in multiple directories. Having your .mJPackage have a different prefix than the .mjCode files in the same directory is likely confusing also.

```

    compilationDirectives {
        prefix "Zoo";
    }

```

There is only one example where we ourselves have used the *prefix* directive: for the Meta-Ja utilites, whose classes have no prefix at all because they are so widely used (e.g. "Strings", not "MjUtilStrings").

```

    compilationDirectives {
        prefix "Zoo";
    }

```

4.20.6 *package*

Rarely used. You can explicitly define the package name for a certain directory as shown below.

```

    compilationDirectives {
        package "edu.isi.dce.mjexamples.basezoo";
    }

```

This is virtually always less desirable than having Meta-Ja compute the right package name for you (see also the *-currentDir* and *-topDir* flags). The only reason you may ever want to do this is because Meta-Ja fails to compute the package name correctly.

4.20.7 *clean*

Rarely used. By adding "clean" to your normal invocation of the Meta-Ja generator, it will not generate any code at all. Instead, it will remove all files that it generated.

You can prevent it from doing so, but we believe the only situation where this is desirable is for the Meta-Ja generator source code itself. Because Meta-Ja is written in Meta-Ja losing the generated Java files as well as all executable class files of it would be catastrophic – I would have to hand-author the Java code for the first boot-strapping by hand to recover, which would take weeks. Thus, I use the following directive in that directory as an extra precaution.

```

    compilationDirectives {
        clean "no";
    }

```

You should likely never use this directive unless you are working on the Meta-Ja generator itself.

4.21 Reference: Meta-Ja Command-Line Arguments

The following arguments are generic to all types of Meta-Ja uses, independent of if you call it to generate Java code, to remove generated Java code, or whatever.

-feedback *full*<default> | *singleLine* | *none* Generic Command-Line Argument
 Controls how much information Meta-Ja tells you about what it is doing. The default is *full*, which will for example tell you the name of every file it writes. *singleLine* only writes a single dot per file written, and *none* will give you no feedback whatsoever unless an error occurs.

-command *generate*<default> | *clean* | *dtd* Generic Command-Line Argument
Controls what the Meta-Ja invocation does. Most often, you will invoke it to generate Java code from the *.mjPackage* and *.mjCode* files in a directory.

4.21.1 Command-Line Arguments for *generate*

The *generate* command generates Java files.

-classpath *//D/Code/somelib.jar* generate Command-Line Argument
The classpath that Meta-Ja uses to read *.mjPackage* file from other directories, Zip files, and JAR files. Meta-Ja will use the value of the CLASSPATH variable if you do not supply this argument. MS-DOS and Windows systems use semicolons to separate multiple entries in a classpath while most other systems use colons. Note that depending on your shell you will have to quote characters such as semicolons and backslashes, typically by either putting single quotes or double quotes around the whole argument, or by preceding those characters with a backward slash.

-currentDir generate Command-Line Argument
//D/Code/source/com/me/project
Tell Meta-Ja what the current directory is. In conjunction with the below, this is used to automatically determine the correct package name for the generated Java files.

-topDir *com* generate Command-Line Argument
Tell Meta-Ja what the root directory for automatic package name determination is. In the examples given, it would determine that the package name should be "com.me.project". Note that this means that you cannot use "com" as another directory name within "com" and have the package name automatically determined. (You can, however, explicitly state the correct package name in a Meta-Ja compilation directive.)

4.21.2 Command-Line Arguments for *clean*

The *clean* command removes previously generated Java files. There are no *clean*-specific further command-line arguments.

4.21.3 Command-Line Arguments for *dtd*

The *dtd* command generates a Document Type Description (DTD) for the automatic XML input/output format that Meta-Ja offers. You can then use a validating XML parser to check if hand-typed XML input for a Meta-Ja-generated class conforms to the DTD (one such parser is *Xerces*, which is freely available from *xml.apache.org*).

-forClass *Animals* dtd Command-Line Argument
Required – specifies the root entity for which to generate the DTD. This is needed because there can be a great number of XML-enabled Meta-Ja entity definitions in a *.mjPackage* file – and Meta-Ja has no way to tell which one is the one for which you want to validate a hand-written XML file. (If you want to generate DTDs for multiple entities, invoke Meta-Ja several times.)

Appendix A The Precise *.mjPackage* Syntax

The file reproduced verbatim below defines the exact syntax of *.mjPackage* files – presuming, of course, you understand how to read the raw JJT parser generator input format. If not, just skip this section; the rest of the document should contain all of the *mjPackage* syntax information you need.

```

SKIP: {<<WhiteSpace>>|<<SingleLineComment>>}
TOKEN:
{ <COMPILATION_DIRECTIVES:"compilationDirectives">
: WithinCompilationDirectives
| <PACKAGE_COMMENT: "/*" (~["*"])* "*" ("*" |
      (~["*","/"] (~["*"])* "*"))* "/">
| <COMMENT: "/*" (~["*"])* "*" ("*" |
      (~["*","/"] (~["*"])* "*"))* "/">
| <OBJECT: "object"> : WithinObjectHeader
| <ABSTRACT: "abstract">
| <ENUMERATION: "enumeration"> : WithinEnumeration
| <INTERFACE: "interface"> : WithinInterface
| <LIST: "list"> : WithinList
| <#SingleLineComment: "//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")>
| <#WhiteSpace: (<WSP>)+ >
| <#WSP: [" ", "\t", "\r", "\n"] >
}

<WithinCompilationDirectives> SKIP: {<<WhiteSpace>>|<<SingleLineComment>>}
<WithinCompilationDirectives> TOKEN:
{ <COMPILATION_OPEN_CURLY: "{">
| <COMPILATION_CLOSE_CURLY: "}"> : DEFAULT
| <DOUBLEQUOTED: "\"" ("\\\"|~["\""])* "\">
| <LNAME: <LLETTER> (<LETTER>|<DIGITORDOT>)*>
| <#LLETTER: ["a"-"z"]>
| <#ULETTER: ["A"-"Z"]>
| <#LETTER: <LLETTER>|<ULETTER>>
| <#DIGITORDOT: ["0"-"9","."]>
| <COMPILATION_SEMICOLON: ";">
}

<WithinObjectHeader> SKIP: {<<WhiteSpace>>|<<SingleLineComment>>}
<WithinObjectHeader> TOKEN:
{ <EXTENDS: "extends">
| <IMPLEMENTS: "implements">
| <PARSED: "-parsed">
| <COMMANDLINE: "-commandLine">
| <CANBECREATEDFROMASTRING: "-canBeCreatedFromAString">
| <XML: "-xml">
| <WRITE_GENERATION_DATE: "-writeGenerationDate">
| <CARRIER: "-carrier">
| <NO_CONSTRUCTORS: "-noConstructors">
}

```

```

| <PRIVATE_CONSTRUCTORS: "-privateConstructors">
| <AUGMENTED_PRIVATE_CONSTRUCTORS: "-augmentedPrivateConstructors">
| <RESTRICTED_CONSTRUCTORS: "-restrictedConstructors">
| <PROTECTED_CONSTRUCTORS: "-protectedConstructors">
| <COPY: "-copy">
| <NO_COPY: "-noCopy">
| <CUSTOM_COPY: "-customCopy">
| <PRINT: "-print">
| <NO_PRINT: "-noPrint">
| <CUSTOM_PRINT: "-customPrint">
| <GEN_FCT: "-genFct">
| <NO_GEN_FCT: "-noGenFct">
| <CUSTOM_GEN_FCT: "-customGenFct">
| <COMMA: ", ">
| <OBJECT_COMMENT: <COMMENT>>
| <OBJECT_OPEN_CURLY: "{"> : WithinObjectBody
| <EXCLAMATION_MARK: "!">
| <CARET: "^">
| <UNAME: <ULETTER> (<LETTER>|<DIGITORDOT>)*>
| <OBJECTHEADER_LNAME: <LNAME>>
}

```

```

<WithinObjectBody> SKIP: {<<WhiteSpace>>|<<SingleLineComment>>}
<WithinObjectBody> TOKEN: {
<OBJECT_CLOSE_CURLY: "}"> : DEFAULT
| <OBJECTBODY_EXCLAMATION_MARK: <EXCLAMATION_MARK>>
| <OBJECTBODY_CARET: <CARET>>
| <COLON: ":">
| <VOLATILE: "-volatile">
| <NOXML: "-noXml">
| <KEY: "-key">
| <POINTER: "-pointer">
| <SET: "-set">
| <VOIDSET: "-voidSet">
| <CUSTOMSET: "-customSet">
| <RESTRICTEDSET: "-restrictedSet">
| <PROTECTEDSET: "-protectedSet">
| <NOGET: "-noGet">
| <ISGET: "-isGet">
| <BAREGET: "-bareGet">
| <RESTRICTEDGET: "-restrictedGet">
| <PROTECTEDGET: "-protectedGet">
| <ISREALLYADATE: "-isReallyADate">
| <CLNOSWITCH: "-clNoSwitch">
| <OBJECT_DOUBLEQUOTED: <DOUBLEQUOTED>>
| <SEMICOLON: ";">
| <PUBLIC: "public">
| <PROTECTED: "protected">
| <PRIVATE: "private">

```

```

| <RESTRICTED:"restricted">
| <STATIC:"static">
| <FINAL:"final">
| <NATIVE:"native">
| <SYNCHRONIZED:"synchronized">
| <VOID:"void">
| <THROWS:"throws">
| <OBJECT_UNAME:<UNAME>>
| <OBJECT_LNAME:<LNAME>>
| <LPARAN:"(">
| <RPARAN:")">
| <OBJECTBODY_COMMENT:<COMMENT>>
| <OBJECTBODY_COMMA:<COMMA>>
}

```

```

<WithinEnumeration> SKIP: {<<WhiteSpace>>|<<SingleLineComment>>}
<WithinEnumeration> TOKEN:
{ <ENUMERATION_UNAME:<UNAME>>
| <ENUMERATION_DOUBLEQUOTED:<DOUBLEQUOTED>>
| <ENUMERATION_COMMA:<COMMA>>
| <ENUMERATION_PARSED: "-parsed">
| <ENUMERATION_XML:<XML>>
| <ENUMERATION_NO_CONSTRUCTORS:<NO_CONSTRUCTORS>>
| <ENUMERATION_PRIVATE_CONSTRUCTORS:<PRIVATE_CONSTRUCTORS>>
| <ENUMERATION_AUGMENTED_PRIVATE_CONSTRUCTORS:<AUGMENTED_PRIVATE_CONSTRUCTORS>>
| <ENUMERATION_RESTRICTED_CONSTRUCTORS:<RESTRICTED_CONSTRUCTORS>>
| <ENUMERATION_PROTECTED_CONSTRUCTORS:<PROTECTED_CONSTRUCTORS>>
| <ENUMERATION_OPEN_CURLY:"{">
| <ENUMERATION_CLOSE_CURLY:"}"> : DEFAULT
| <ENUMERATION_COMMENT:<COMMENT>>
}

```

```

<WithinInterface> SKIP: {<<WhiteSpace>>|<<SingleLineComment>>}
<WithinInterface> TOKEN:
{ <INTERFACE_EXTENDS:<EXTENDS>>
| <INTERFACE_XML:<XML>>
| <INTERFACE_COPY:"-copy">
| <INTERFACE_PRINT:"-print">
| <INTERFACE_GEN_FCT:"-genFct">
| <INTERFACE_UNAME:<UNAME>>
| <INTERFACE_EXCLAMATION_MARK:<EXCLAMATION_MARK>>
| <INTERFACE_CARET:<CARET>>
| <INTERFACE_COMMA:<COMMA>>
| <INTERFACE_CARRIER:<CARRIER>>
| <INTERFACE_OPEN_CURLY:"{">
| <INTERFACE_CLOSE_CURLY:"}"> : DEFAULT
| <INTERFACE_COMMENT:<COMMENT>>
| <INTERFACE_SEMICOLON:<SEMICOLON>>
| <INTERFACE_PUBLIC:<PUBLIC>>
}

```

```

| <INTERFACE_PROTECTED:<PROTECTED>>
| <INTERFACE_PRIVATE:<PRIVATE>>
| <INTERFACE_RESTRICTED:<RESTRICTED>>
| <INTERFACE_STATIC:<STATIC>>
| <INTERFACE_FINAL:<FINAL>>
| <INTERFACE_NATIVE:<NATIVE>>
| <INTERFACE_SYNCHRONIZED:<SYNCHRONIZED>>
| <INTERFACE_VOID:<VOID>>
| <INTERFACE_LPARAN:<LPARAN>>
| <INTERFACE_RPARAN:<RPARAN>>
| <INTERFACE_THROWS:<THROWS>>
| <INTERFACE_LNAME:<LNAME>>
}

<WithinList> SKIP: {<<WhiteSpace>>|<<SingleLineComment>>}
<WithinList> TOKEN:
{ <LIST_UNAME:<UNAME>>
| <LIST_LNAME:<LNAME>>
| <LIST_EXCLAMATION_MARK:<EXCLAMATION_MARK>>
| <LIST_CARET:<CARET>>
| <LIST_COLON:<COLON>>
| <LIST_OPEN_ANGULAR:"<">
| <LIST_CLOSE_ANGULAR:">">
| <LIST_OPEN_RECTANGULAR:"["> : WithinListKeyArgs
| <LIST_PARSED:<PARSED>>
| <LIST_XML:<XML>>
| <LIST_NO_CONSTRUCTORS:<NO_CONSTRUCTORS>>
| <LIST_PRIVATE_CONSTRUCTORS:<PRIVATE_CONSTRUCTORS>>
| <LIST_AUGMENTED_PRIVATE_CONSTRUCTORS:<AUGMENTED_PRIVATE_CONSTRUCTORS>>
| <LIST_RESTRICTED_CONSTRUCTORS:<RESTRICTED_CONSTRUCTORS>>
| <LIST_PROTECTED_CONSTRUCTORS:<PROTECTED_CONSTRUCTORS>>
| <LIST_DATABASE_TABLE:"-databaseTable">
| <LIST_CARRIER:<CARRIER>>
| <LIST_NO_COPY:<NO_COPY>>
| <LIST_CUSTOM_COPY:<CUSTOM_COPY>>
| <LIST_NO_PRINT:<NO_PRINT>>
| <LIST_CUSTOM_PRINT:<CUSTOM_PRINT>>
| <LIST_NO_GEN_FCT:<NO_GEN_FCT>>
| <LIST_CUSTOM_GEN_FCT:<CUSTOM_GEN_FCT>>
| <LIST_KEY:"-key">
| <LIST_OPEN_CURLY:"{">
| <LIST_CLOSE_CURLY:"}"> : DEFAULT
| <LIST_COMMENT:<COMMENT>>
}

<WithinListKeyArgs> TOKEN:
{ <LISTKEYARGS_COMMA:<COMMA>>
| <LISTKEYARGS_CLOSE_RECTANGULAR:"]"> : WithinList
| <LISTKEYARGS_UNIQUE:"unique">

```

```

| <LISTKEYARGS_MULTIPLEMATCHESPOSSIBLE:"multipleMatchesPossible">
| <LISTKEYARGS_PRIMARY:"primary">
| <LISTKEYARGS_SECONDARY:"secondary">
| <LISTKEYARGS_HASHED:"hashed">
| <LISTKEYARGS_NOTHASHED:"notHashed">
| <LISTKEYARGS_TYPE:"type">
| <LISTKEYARGS_EQUALS:"="> : WithinListKeyArgsType
}

```

```

<WithinListKeyArgsType> TOKEN:
{
<LISTKEYARGS_EXCLAMATION_MARK:<EXCLAMATION_MARK>>
| <LISTKEYARGS_CARET:<CARET>>
| <LISTKEYARGS_UNAME:<UNAME>> : WithinListKeyArgs
| <LISTKEYARGS_LNAME:<LNAME>> : WithinListKeyArgs }

```

```

MjPackage Package() :
{ [PackageComment() #PrshlpPackageComment(1)]
  [CompilationDirectives()]
  Entities()
  <EOF>
}

void CompilationDirectives() :
{ <COMPILATION_DIRECTIVES> <COMPILATION_OPEN_CURLY>
  (CompilationDirective()* <COMPILATION_CLOSE_CURLY>)}

void CompilationDirective() :
{ LowercaseIdentifier() DoubleQuotedString()
  [<COMPILATION_SEMICOLON>]}

void Entities() : {(Entity()*}
void Entity() #void : {Object()|Enumeration()|List()|Interface()}

void Enumeration() :
{<ENUMERATION> UppercaseIdentifier()
  [<ENUMERATION_PARSED> #PrshlpParsed]
  [<ENUMERATION_XML> #PrshlpXml]
  [<ENUMERATION_NO_CONSTRUCTORS> #PrshlpNoConstructors]
  [<ENUMERATION_PRIVATE_CONSTRUCTORS> #PrshlpPrivateConstructors]
  [<ENUMERATION_AUGMENTED_PRIVATE_CONSTRUCTORS> #PrshlpAugmentedPrivateConstructors]
  [<ENUMERATION_RESTRICTED_CONSTRUCTORS> #PrshlpRestrictedConstructors]
  [<ENUMERATION_PROTECTED_CONSTRUCTORS> #PrshlpProtectedConstructors]
  [Comment() #PrshlpComment(1)]
  <ENUMERATION_OPEN_CURLY>
  EnumerationValues()
  <ENUMERATION_CLOSE_CURLY>}

```

```

void EnumerationValues() :
{EnumerationValue() (<ENUMERATION_COMMA> EnumerationValue())*}
void EnumerationValue() :
{DoubleQuotedString() [Comment() #PrshlpComment(1)]}

void TypePotentiallyIncludingAColon() #Type :
  {[ExclamationMark() #PrshlpAbsoluteWithinMj |
  Caret() #PrshlpAbsoluteOutsideMj]
  Identifier()
  [Colon() Identifier() #PrshlpColonBase(1)]}
void Type() :
  {[ExclamationMark() #PrshlpAbsoluteWithinMj |
  Caret() #PrshlpAbsoluteOutsideMj]
  Identifier() }
void Object() :
  {[<ABSTRACT> #PrshlpAbstractIndicator]
  <OBJECT> UppercaseIdentifier()
  ObjectDirectives()
  [Comment() #PrshlpComment(1)]
  <OBJECT_OPEN_CURLY>
  VariableDeclarations() MethodDeclarations()
  <OBJECT_CLOSE_CURLY>}
void ObjectDirectives() #void : {(ObjectDirective())*}
void ObjectDirective() #void : {
  <EXTENDS> Type() #PrshlpExtends(1) |
  <IMPLEMENTS> Type()(<COMMA>Type())* |
  <PARSED> #PrshlpParsed |
  <COMMANDLINE> #PrshlpCommandLine |
  <CANBECREATEDFROMASTRING> #PrshlpCanBeCreatedFromAString |
  <XML> #PrshlpXml |
  <WRITE_GENERATION_DATE> #PrshlpWriteGenerationDate |
  <NO_CONSTRUCTORS> #PrshlpNoConstructors |
  <PRIVATE_CONSTRUCTORS> #PrshlpPrivateConstructors |
  <AUGMENTED_PRIVATE_CONSTRUCTORS> #PrshlpAugmentedPrivateConstruc-
tors |
  <RESTRICTED_CONSTRUCTORS> #PrshlpRestrictedConstructors |
  <PROTECTED_CONSTRUCTORS> #PrshlpProtectedConstructors |
  <COPY> #PrshlpCopy |
  <NO_COPY> #PrshlpNoCopy |
  <CUSTOM_COPY> #PrshlpCustomCopy |
  <PRINT> #PrshlpPrint |
  <NO_PRINT> #PrshlpNoPrint |
  <CUSTOM_PRINT> #PrshlpCustomPrint |
  <GEN_FCT> #PrshlpGenFct |
  <NO_GEN_FCT> #PrshlpNoGenFct |
  <CUSTOM_GEN_FCT> #PrshlpCustomGenFct |
  <CARRIER> #PrshlpCarrier
}
void VariableDeclarations() : {(VariableDeclaration())*}

```

```

void VariableDeclaration() : {
    TypePotentiallyIncludingAColon()
    [LowercaseIdentifier()]
    VariableDeclarationDirectives()
    [Comment() #PrshlpComment(1)]
    <SEMICOLON>}
void VariableDeclarationDirectives() #void :
{(VariableDeclarationDirective()*}
void VariableDeclarationDirective() #void : {
    <VOLATILE> #PrshlpVolatile |
    <NOXML> #PrshlpNoXml |
    <KEY> #PrshlpKey |
    <POINTER> #PrshlpPointer |
    <SET> #PrshlpSet |
    <VOIDSET> #PrshlpVoidSet |
    <CUSTOMSET> DoubleQuotedString() #PrshlpCustomSet(1) |
    <RESTRICTEDSET> #PrshlpRestrictedSet |
    <PROTECTEDSET> #PrshlpProtectedSet |
    <NOGET> #PrshlpNoGet |
    <ISGET> #PrshlpIsGet |
    <BAREGET> #PrshlpBareGet |
    <RESTRICTEDGET> #PrshlpRestrictedGet |
    <PROTECTEDGET> #PrshlpProtectedGet |
    <ISREALLYADATE> #PrshlpIsReallyADate |
    <CLNOSWITCH> #PrshlpClNoSwitch |
    DoubleQuotedString() #PrshlpVariableDefault(1)
}

void Interface() :
{<INTERFACE> UppercaseIdentifier() InterfaceDirectives()
 [Comment() #PrshlpComment(1)]
 <INTERFACE_OPEN_CURLY>
 MethodDeclarations()
 <INTERFACE_CLOSE_CURLY>}
void InterfaceDirectives() #void : {(InterfaceDirective()*}
void InterfaceDirective() #void : {
    <INTERFACE_EXTENDS> Type()(<INTERFACE_COMMA>Type()* |
    <INTERFACE_COPY> #PrshlpCopy |
    <INTERFACE_XML> #PrshlpXml |
    <INTERFACE_PRINT> #PrshlpPrint |
    <INTERFACE_GEN_FCT> #PrshlpGenFct |
    <INTERFACE_CARRIER> #PrshlpCarrier}

void List() :
{<LIST>
 <LIST_OPEN_ANGULAR>
 TypePotentiallyIncludingAColon()
 <LIST_CLOSE_ANGULAR>
 [<LIST_OPEN_ANGULAR>

```

```

    Type() #PrshlpListType(1)
    <LIST_CLOSE_ANGULAR>]
  [UppercaseIdentifier()] ListDirectives()
  [Comment() #PrshlpComment(1)]
  <LIST_OPEN_CURLY> <LIST_CLOSE_CURLY> }
void ListDirectives() #void : {(ListDirective())* }
void ListDirective() #void : {
  <LIST_PARSED> #PrshlpParsed |
  <LIST_XML> #PrshlpXml |
  <LIST_NO_CONSTRUCTORS> #PrshlpNoConstructors |
  <LIST_PRIVATE_CONSTRUCTORS> #PrshlpPrivateConstructors |
  <LIST_AUGMENTED_PRIVATE_CONSTRUCTORS> #PrshlpAugmentedPrivateConstruc-
tors |
  <LIST_RESTRICTED_CONSTRUCTORS> #PrshlpRestrictedConstructors |
  <LIST_PROTECTED_CONSTRUCTORS> #PrshlpProtectedConstructors |
  <LIST_CARRIER> #PrshlpCarrier |
  KeyDeclaration() |
  <LIST_NO_COPY> #PrshlpNoCopy |
  <LIST_CUSTOM_COPY> #PrshlpCustomCopy |
  <LIST_NO_PRINT> #PrshlpNoPrint |
  <LIST_CUSTOM_PRINT> #PrshlpCustomPrint |
  <LIST_NO_GEN_FCT> #PrshlpNoGenFct |
  <LIST_CUSTOM_GEN_FCT> #PrshlpCustomGenFct |
  <LIST_DATABASE_TABLE> #PrshlpDatabaseTable
}
void KeyDeclaration() : {
  <LIST_KEY>
  [<LIST_OPEN_RECTANGULAR>
  KeyDirectives()
  <LISTKEYARGS_CLOSE_RECTANGULAR>]
  LowercaseIdentifier() }
void KeyDirectives() #void : {
  KeyDirective() (<LISTKEYARGS_COMMA> KeyDirective())* }
void KeyDirective() #void : {
  <LISTKEYARGS_UNIQUE> #PrshlpUnique |
  <LISTKEYARGS_MULTIPLEMATCHESPOSSIBLE>
  #PrshlpMultipleMatchesPossible |
  <LISTKEYARGS_PRIMARY> #PrshlpPrimary |
  <LISTKEYARGS_SECONDARY> #PrshlpSecondary |
  <LISTKEYARGS_HASHED> #PrshlpHashed |
  <LISTKEYARGS_NOTHASHED> #PrshlpNotHashed |
  <LISTKEYARGS_TYPE> <LISTKEYARGS_EQUALS> Type() }

void MethodDeclarations() : {(MethodDeclaration())* }
void MethodDeclaration() : {
  MethodAccess() (MethodModifier())* MethodReturnType()
  [Comment() #PrshlpReturnTypeComment(1)]
  Identifier() MethodParameters() [Throws() MethodTypeList()]

```

```

    [Comment() #PrshlpMethodComment(1)] Semicolon() }
void MethodAccess() #void :
{(Public()|Protected()|Private()|Restricted())
 #PrshlpMethodAccess(1)}
void MethodModifier() #void :
{(Static()|Final()|Native()|Synchronized())
 #PrshlpMethodModifier(1)}
void MethodReturnType() #void :
{Void()|(Type() #PrshlpReturnType(1))}
void MethodParameters() :
{Lparan()
 [ MethodParameter() ( Comma() MethodParameter() )* ]
 Rparan()}
void MethodParameter() :
{Type() [LowercaseIdentifier()] [Comment() #PrshlpComment(1)]}
void MethodTypeList() #void : {Type() (Comma() Type())*}

void ExclamationMark() #void :
{ <EXCLAMATION_MARK>|<OBJECTBODY_EXCLAMATION_MARK>|
 <LIST_EXCLAMATION_MARK>|<LISTKEYARGS_EXCLAMATION_MARK>|
 <INTERFACE_EXCLAMATION_MARK>}
void Caret() #void :
{ <CARET>|<OBJECTBODY_CARET>|
 <LIST_CARET>|<LISTKEYARGS_CARET>|
 <INTERFACE_CARET>}
void Colon() #void :
{ <COLON> | <LIST_COLON> }
void Semicolon() #void : {<SEMICOLON>|<INTERFACE_SEMICOLON>}
void Public() #StringMm :
|
 <INTERFACE_PUBLIC>}
void Protected() #StringMm :
{(<PROTECTED>|<INTERFACE_PROTECTED>)
 }
void Private() #StringMm :
{(<PRIVATE>|<INTERFACE_PRIVATE>)
 }
void Restricted() #StringMm :
{(<RESTRICTED>|<INTERFACE_RESTRICTED>)
 }
void Static() #StringMm :
{(<STATIC>|<INTERFACE_STATIC>)
 }
void Final() #StringMm :
{(<FINAL>|<INTERFACE_FINAL>)
 }
void Native() #StringMm :
{(<NATIVE>|<INTERFACE_NATIVE>)
 }

```

```

    }
void Synchronized() #StringMm :
{(<SYNCHRONIZED>|<INTERFACE_SYNCHRONIZED>)
  }
void Void() #void :
{<VOID>|<INTERFACE_VOID>}
void Throws() #void :
{<THROWS>|<INTERFACE_THROWS>}
void Lparan() #void :
{<LPARAN>|<INTERFACE_LPARAN>}
void Comma() #void :
{<COMMA>|<OBJECTBODY_COMMA>|<INTERFACE_COMMA>|<ENUMERATION_COMMA>}
void Rparan() #void : {<RPARAN>|<INTERFACE_RPARAN>}

void Identifier() #void :
{ UppercaseIdentifier() | LowercaseIdentifier() }
void LowercaseIdentifier() #StringMm :
|
  <OBJECTHEADER_LNAME> |
  <OBJECT_LNAME> |
  <LIST_LNAME> |
  <LISTKEYARGS_LNAME> |
  <INTERFACE_LNAME> }
void UppercaseIdentifier() #StringMm :
|
  <OBJECT_UNAME> |
  <LIST_UNAME> |
  <ENUMERATION_UNAME> |
  <LISTKEYARGS_UNAME> |
  <INTERFACE_UNAME> }
void Comment() #StringMm :
|
  <OBJECT_COMMENT> |
  <OBJECTBODY_COMMENT> |
  <LIST_COMMENT> |
  <ENUMERATION_COMMENT> |
  <INTERFACE_COMMENT> }

void PackageComment() #StringMm :
|
  <COMMENT> }

void DoubleQuotedString() #StringMm :

| <ENUMERATION_DOUBLEQUOTED>
| <OBJECT_DOUBLEQUOTED> }

void DoubleQuotedStrings() #void :

```

```
{(DoubleQuotedString()*}  
void CommaSeparatedDoubleQuotedStrings() #void :  
  {[DoubleQuotedString()(Comma() DoubleQuotedString()*]}
```

Appendix B Java Coding Tips

This section describes the author's personal coding practices. These are described for the reader's general benefit and do not present a requirement for using the Meta-Ja generator (but the generator is designed to work well with these practices). Understanding my coding practices will also help in studying the Mastermind source code.

Use unique class names. For example, if you have an `MyObject` class in both package `Xxx` and package `Yyy` you may be tempted to call it `such`. The problem is that you then end up having to use full-length prefixes to disambiguate which class you are referring to in downstream code ("`edu.isi.mastermind.Xxx.Object`" and "`edu.isi.mastermind.Yyy.Object`", for example) - Java doesn't offer relative prefixing such as "`Xxx.Object`" and "`Yyy.Object`". Furthermore you can easily get confused which of the two `Object.mjCode` files you are currently editing. For all of the above reasons, these classes would be named `XxxObject` and `YyyObject` - uglier and reminiscent of the C++ style but worth it.

Use a unique prefix for every package, and name your .mjPackage file after it. For example, I would make up a "`ZaUi`" prefix for the "`UserInterface`" package within the "`ZooAnimals`" project, and call my package description file `ZaUi.mjPackage`.

Don't sub-class non-abstract classes, ever. This is because a non-abstract class by definition has to implement all inherited abstract functions. That means that if you want to further override an abstract function in the derived class but forget to (or mis-type the function name by even one character) the Java *run-time* interpreter will call the wrong function - and you'll be sitting there debugging for a while. In contrast, having leaf concrete classes implement the same abstract function will yield a *design-time* error that is easily addressed.

Don't call a function from both inside and outside a class. For example, say that you have a `computeLayout(Element e)` function on class `X`, and that the function recursively calls itself. In this case, I would make up a private `computeLayoutInternal()` functions that calls `computeLayoutInternal()` recursively, and a public `computeLayout()` function that just calls the internal one. There are many good reasons: you may want to trace external calls to your functions (and not be confused by apparently external calls that are really internal), you may want to *synchronize* the external calls (but not every internal one), and so on...

No representation without computation. (Slogan credit: Drew McDermott) If you're coding by yourself, don't add data attributes which are un-used, rather add the computation on it simultaneously with the attribute. Obviously, this doesn't apply when you're prototyping a data structure for agreement with others.

Use different packages for input, output, and computation . This obviously only applies for major computations, typically these that allocated intermediate data structures on the way to the final output.

Appendix C Unimplemented Ideas

C.1 Down-cast Set Functions on Derived Classes

Imagine that you have a class *Animal* with attributes *name* and *age*, and a derived class *Giraffe* with attribute *lengthOfNeck*. It is convenient to be able to call the following sequence of set functions without having to use temporary variables:

```
... = new Giraffe("susan").setAge(2).setLengthOfNeck(30);
```

The problem, of course, is that you can't because the return type of the *setAge()* call is of type *Animal*, so that the subsequent call of *setLengthOfNeck()* fails. The solution is in the automatically implemented function *setGiraffeAge()* on *Giraffe* that is of return type *Giraffe*. That is, you can say:

```
... = new Giraffe("susan").setGiraffeAge(2).setLengthOfNeck(30);
```

C.2 -key and -pointer declarations

Sometimes, you want to have "pointer" data members. For example, your zoo database may have a "keeper" data member for animals that points to a human in the zoo database that is supposed to take care of it (it points in the sense that the human will not be removed from the database just because the animal was removed - she may take care of many other animals simultaneously). In that case, you would declare the keeper data member a *-pointer* and identify a String-valued attribute the *-key* of the employee:

```
object Employee {
    String socialSecurityNumber -key;
    String lastName;
}
object Animal {
    String name;
    Employee careTaker -pointer;
}
```

(Note that only up to one attribute of an object can be declared *-key*; this presents a current implementation limitation of the Meta-Ja generator. It is desirable in principle to be able to declare multi-valued keys, and to declare alternative additional keys, but that is not implemented.)

When the XML for *Animal* is written out, just the key for the caretaker employee is written out (not e.g. the employee's salary and home phone number). Conversely, when the XML is read back in, you must pass a mechanism TBD which allows the XML-reading function to call "public Object locate(String type,String key)" so it can translate the symbolic key back to an *Employee* memory pointer.

(Note that that strategy implies that you should read all the employees first, and the animals later; the current implementation cannot handle circular dependencies - the case where animals point at employees *and* employees point to animals - because it is a single-pass parser. However, such circular data schema designs are rarely a good idea in the first place...)

Appendix D Open To-Do Items

support hashing of non-String types (entered *2001-04-13*)

Especially integers, and object memory pointers. **document the ^ and . type modifiers** (entered *2001-03-21*)

I emailed that info out but it is not in the Meta-Ja reference manual. **flush mjl.x and each individual package** (entered *2001-03-21*)

Right now Meta-Ja seems to hang forever, then prints its whole first line at once. **multipleMatchesPossible should imply notHashed** (entered *2001-03-21*)

Meta-Ja list support automatic implementation of hashing on a single key for unique lists, but not for non-unique lists. Saying both "hashed" and "multipleMatchesPossible" results in writing out invalid code. **Show a + in front of XML imports** (entered *2001-03-21*)

Otherwise, the printout of included packages can be misleading, for example I thought there were circular imports in David's trm structure to the SNAP code, but they turned out to be just xmlimports. **convert VariableNameTreatment warnings to errors** (entered *2001-03-21*)

Right now, saying e.g. !JFrame rather than ^JFrame will issue a warning but keep Meta-Ja going. Should become an error that stops MJ in a few weeks once everyone has gotten used to the new style. **listinterface<X>** (entered *2001-03-16*)

List constructs for API-only directories. Currently, you either get the choice of using untyped Iterator as the list construct in API directories, or to use concrete list<X> constructs (and hence make the directory not all-interface anymore).

The new listinterface construct will be a better solution: in the API package: interface X {} listinterface<X> {} in the implementation package: object Y implements X {} list<Y> implements Xs {}

The generated list functions for listinterface<X> will have the expanded type name in the function names for all functions that return non-void, such as "ApiX popApiXs()". This is ugly but necessary because Java does not permit overloading by return type. **customConstructors flag** (entered *2001-03-16*)

With that flag, MJ writes commented-out constructor code and writes code for e.g. copying and XML as usual. Thus, users can provide their own constructors for cacheing, database initialization, etc., and they can look at the commented-out ones in the .java file for inspiration. **require ! in front of every unknown identifier** (entered *2001-03-16*)

It seems some still slip through without it (especially in "extends" clauses?). This is awful because if you refer to class Yyy of prefix XX as "XxYyy" (rather than as either "Yyy" or "!XxYyy") hand-written Java code will still work but Meta-Ja will not e.g. recognize that the class is XML-capable – and finding this bug is difficult and time-consuming. **enforce consistent import order** (entered *2001-03-16*)

The idea is that if the inclusion order in any later .mjPackage file differs from the one in an earlier .mjPackage file MJ complains and stops. This will make users more aware of the concept of a defined compilation order; obviously this would necessitate me going through a lot of existing code to straighten out the existing compilation orders but it would still be worthwhile. **check for duplicate imports** (entered *2001-03-16*)

Jinbo produced a weird bug where Meta-Ja failed to infer that a class was XML-capable because its .mjPackage was included twice and not next to each other. **separate shorter XML identifiers** (entered *2001-02-27*)

Use an `-xmlname` tag to provide XML names - that way they can use descriptive names in memory and short names in the file format. For example:

```
object ComplexTask -xmlname CT {Resource:Capability -xmlname c;} warn on
xmlimport of non-existing packages (entered 2001-02-27)
```

Apparently that is silently ignored at the moment. **immutable and mutable iterator APIs** (entered 2001-02-27)

Those would be generated by something like "abstract list<Item> {}" and "abstract list<Item> -immutable {}". They can then be used in interface-only API definition packages where we now use untyped Iterators. **warn about missing exclamation mark** (entered 2001-02-27)

For example, "object PilotTrainingManual extends RmSupSimpleCapabilityDimensionSet" did not warn about the missing exclamation mark in front of RmSup. **better -help messages** (entered 2001-02-21)

Print the comment text for the attributes. Would also be nice to print the possibilities for an enumeration, and have some mapping to texinfo for automatic import into documentation. **DTD support** (entered 2001-02-21)

Finish the command-line reform effort and DTD support option. **Expose generated enumeration constants** (entered 2001-02-12)

That way, one can refer to `sXxx` rather than having to call `createXxx()`, saving the function call overhead. The original reason not to allow that was because there used to be a `set()` function - so a hapless user could call `ThreeValuedTruth.sTrue.set(ThreeValuedTruth.sFalse)` - changing the meaning of the constants! Enumerations are now "immutable" so the latter is no longer a concern. **Tolerate XML comments** (entered 2001-01-31)

Meta-Ja should ignore comments like `<!-- ... -->` without printing warning lines about them. **Tolerate XML version line** (entered 2001-01-31)

Meta-Ja should be able to ignore the `<?xml version="1.0" encoding="UTF-8"?>` tag, and indeed all `<? ... ?>` processing instructions. **Interpret lack of XML tags as empty list** (entered 2001-01-31)

At the moment, if your object X has a list-valued non-optional attribute L then XML instances must carry the tag, such as:

```
<X><N>my name</N><L></L></X>
```

It should be legal to omit the empty L tags, like this:

```
<X><N>my name</N></X>
```

which would result into the exact same in-memory instance. Note that the treatment is then different for optional and required list-valued variables: missing optional ones will be initialized to NULL but missing required ones to the empty list. **Tolerate XML singular tags** (entered 2001-01-31)

If tools other than Meta-Ja itself produce XML, they reduce tags such as `<l></l>` to the singular tag `<l/>`. Meta-Ja should be able to handle those - it ignores them with a warning message right now. **Provide enumeration values in -help** (entered 2001-01-30)

Meta-Ja command-line support already provides a list of possible command-line arguments when you say "-help" in a command-line, but it does not list the values of enumerations. It should - or otherwise you have to hunt down the `.mjPackage` file to find out what they are! (This came up in conjunction with David's re-organization of SNAP .bat files where it would also be nice to have the enumeration values shown in a comment line). **Honor**

-pointer when printing (entered *2001-01-30*)

Pedro writes: What is the effect of -pointer on an object definition? One thing that it should do is to cause the print method of the containing object to not call print recursively on the pointer (but just print the pointer itself). It seems to me that one common use of -pointer is to include back pointers in data structures, for examples parents in a tree. In that case the print behavior would be very handy. **Address pitfall of having to say -copy for abstract classes** (entered *2001-01-18*)

Not defining it for an abstract class and then being confused when working with the classes later is an eternal pitfall. **Document the pointer/key feature idea** (entered *2001-01-16*)

Most of the time a Java class "owns" data members in the sense that they can be deallocated if it itself is deallocated, but sometimes it just "points" to external data that will persist. The idea is that by marking data members pointed to with "-pointer" the written-out XML will only write out the "key" of that data member (such as a unique name) - not the entire data structure as is the case now. **Flag for turning off the "Deleting and Re-Writing" messages** (entered *2001-01-16*)

I'm getting annoyed by them because they make you scroll down in the Emacs compilation buffer if you "M-x compile" in any but the most trivial directories.. **Clean up BasicList set functions** (entered *2001-01-16*)

The functions for set union, intersection, and subtraction should be static - right now they are member functions on a list that is supposed to be freshly initialized. (This design goes five years back when the functions were written in C++ using templates.) **Private auto-written constructors** (entered *2001-01-16*)

David writes: I suggest a flag that instructs Meta-Ja to make all the constructors on an object private. This forces the use of a factory method by outsiders. (entered *2000-11-28*)

All data members are written out as protected but there is a case for making them private so not even classes in the same package or subclasses can get to them. (entered *2000-09-18*)

You should be able to say "import x.y.z;" without `_requiring_` double quotes around the "x.y.z". A semicolon `_should_` be required to be more in line with Java conventions in the `mjPackage` syntax, and to exploit Emacs' JDE-mode automatic indentation.

Appendix E Completed To-Do Items

require ! in front of every unknown identifier (entered 2001-03-16, closed 2001-03-21)

Description: It seems some still slip through without it (especially in "extends" clauses?). This is awful because if you refer to class Yyy of prefix XX as "XxYyy" (rather than as either "Yyy" or "!XxYyy") hand-written Java code will still work but Meta-Ja will not e.g. recognize that the class is XML-capable – and finding this bug is difficult and time-consuming.

Closure remark: Done - introduced the new ^ type modifier in addition to ! (MJ-unknown reference vs. MJ-known non-local reference).

warn about missing exclamation mark (entered 2001-02-27, closed 2001-03-21)

Description: For example, "object PilotTrainingManual extends RmSupSimpleCapabilityDimensionSet" did not warn about the missing exclamation mark in front of RmSup.

Closure remark: Done - introduced the new ^ type modifier in addition to ! (MJ-unknown reference vs. MJ-known non-local reference).

DTD support (entered 2001-02-21, closed 2001-03-21)

Description: Finish the command-line reform effort and DTD support option.

Closure remark: Meta-Ja now uses Meta-Ja command-line parsing support for itself. There is now a simple DTD generation option it is tested in the mjexamples package and documented as well.

Tolerate XML singular tags (entered 2001-01-31, closed 2001-02-06)

Description: If tools other than Meta-Ja itself produce XML, they reduce tags such as <l></l> to the singular tag <l/>. Meta-Ja should be able to handle those - it ignores them with a warning message right now.

Interpret lack of XML tags as empty list (entered 2001-01-31, closed 2001-02-06)

Description: At the moment, if your object X has a list-valued non-optional attribute L then XML instances must carry the tag, such as:

```
<X><N>my name</N><L></L></X>
```

It should be legal to omit the empty L tags, like this:

```
<X><N>my name</N></X>
```

which would result into the exact same in-memory instance. Note that the treatment is then different for optional and required list-valued variables: missing optional ones will be initialized to NULL but missing required ones to the empty list.

Tolerate XML comments (entered 2001-01-31, closed 2001-02-06)

Description: Meta-Ja should ignore comments like <!-- ... --> without printing warning lines about them.

Tolerate XML version line (entered 2001-01-31, closed 2001-02-06)

Description: Meta-Ja should be able to ignore the `<?xml version="1.0" encoding="UTF-8"?>` tag, and indeed all `<? ... ?>` processing instructions.

Document the pointer/key feature idea (entered *2001-01-16*, closed *2001-02-06*)

Description: Most of the time a Java class "owns" data members in the sense that they can be deallocated if it itself is deallocated, but sometimes it just "points" to external data that will persist. The idea is that by marking data members pointed to with "-pointer" the written-out XML will only write out the "key" of that data member (such as a unique name) - not the entire data structure as is the case now.

Flag for turning off the "Deleting and Re-Writing" messages (entered *2001-01-16*, closed *2001-03-21*)

Description: I'm getting annoyed by them because they make you scroll down in the Emacs compilation buffer if you "M-x compile" in any but the most trivial directories..

Closure remark: There now is a "-feedback singleLine" option (in addition to the default "-feedback full" and the "-feedback none" option).