Programming Language MixJuice 1.0
Users Manual
(DRAFT)

Yuuji Ichisugi

y-ichisugi@aist.go.jp
http://staff.aist.go.jp/y-ichisugi/

National Institute of Advanced Industrial Science and Technology
/ Japan Science and Technology Corporation

May 21, 2002

Abstract

MixJuice(abbreviated as *MJ*) is an enhancement of the Java language which adopts *difference-based modules* instead of Java's original module mechanism. We have completely separated the class mechanism and the module mechanism, and then unified the module mechanism and the differential programming mechanism. This module mechanism enhances the extensibility, reusability and maintainability of programs. In particular, collaborations, which crosscut several classes, can be separated into different modules that can be developed and tested by independent development teams.

The language features of MJ can be extended by adding *EPP plug-ins*. *EPP* (Extensible Pre-Processor) is a language extension framework, which are used to implement the MJ language. Currently, *Collection* plug-in and *assert2* plug-in are included in the MJ language by default. Collection plug-in provides built-in parameterized collection types and `foreach` statement. assert2 plug-in is an emulation of Java2 1.4 assert statement.

Contents

Chapter 1

Design principles of MJ

1.1 The principle of difference definition

Difference-based modules, the module mechanism of MJ, are based on the following design principle.

> **The principle of difference definition:** A module is the difference between the original program and the extended program. The difference is a set of definitions of new names and modifications of definitions of existing names.

Modules are units of reuse, information hiding and separate compilation. The executable application is constructed by linking of modules. In the case of difference-based modules, linking of modules means adding all differences defined by the modules to the empty program.

Difference-based modules can be applied to various programming languages. In many programming languages, a program consists of names and their definitions. For example, in the case of imperative languages, a program is a set of definitions of procedures and data structures. In the case of Java, a program is a set of definitions of classes, fields and methods. The MJ language is a modified Java language, which adopts difference-based modules instead of Java's original module mechanism. In other words, in MJ, a module is a set of additions and modifications of classes, fields and methods.

Modules may inherit other modules. In MJ, both the module-inheritance mechanism and the traditional class-inheritance mechanism can be independently available. Class inheritance and module inheritance are different, as described next. Class inheritance is a mechanism for describing the difference between classes. Module Inheritance is a mechanism for describing the difference between two programs consisting of one or more classes. Using class inheritance, the programmers can only define a new class which has a different name from that of the original class. By module inheritance, the programmers can modify the definitions of existing classes and methods without changing their names. Class inheritance is a mechanism for subtyping and safe late binding. Module inheritance is a mechanism for static reuse and information hiding.

Classes no longer have the functions of modules. In other words, classes are no longer units of reuse, information hiding, or separate compilation.

Difference-based modules have the following merits compared with traditional class-based modules.

- **High extensibility of applications**
  It is easy to write highly extensible applications. There are two reasons for this. One is that all class and method names act as "hooks" for programmers of extension modules. The other reason is that each extension module is composable as a mixin, using multiple inheritance of modules.

- **Class-independency of units of reuse**
  Programmers can define the units of reuse completely independently of boundaries of classes. The programmers can make codes that crosscut some classes, namely collaborations, units of reuse.

- **Extensibility by third party programmers**
  Third party programmers can provide extension modules to extend existing applications. The programmers do not need to have the source-code of the original programs.

- **Module-composability by end-users**
  End users can compose existing modules that provide selected functions to create their own customized applications. The composition of modules does not require any lines of "glue code". It only requires a set of module names.

- **Flexibility of module grouping**
  The programmers can make groups of modules and give names to them to simplify their use. In the case of Java, a certain degree of grouping is possible due to the package mechanism and the use of an "`import`" declaration in the form of "`import p.*;`". For difference-based modules, however, more flexible grouping is possible.

1.2 The principle of name space inheritance

The module mechanism of MJ is based on the following design principle concerning information hiding.

> **The principle of name space inheritance:** All names that are defined at a module are visible from the module itself and its descendant modules, and are invisible from the other modules.

More specifically, "names" means the class, field and method names. The module mechanism of MJ enables more flexible name space management than that of Java by means of this simple rule concerning visibility.

Classes are no longer the units of information hiding in the source-code. All fields in a class are accessible from the defining module and the descendant modules of the module, even if the accessor class is different from the owner of the fields. In MJ, there are no access modifiers (`public`, `protected` or `private`), package mechanisms or nested class mechanisms[1].

Difference-based modules have the following advantages with respect to information hiding compared with Java.

- **Class-independency of units of information hiding**
  Programmers can make the boundaries of information hiding independent of class boundaries. For example, programmers can make collaboration units of information hiding. In addition, to improve the maintainability of the source-code, a programmer can minimize the size of the name space on which their source-code depends. This is especially effective if the number of functions of classes increases and the size of the classes thus becomes bigger and bigger.

- **Flexibility of name space structures**
  The name spaces can form nested structures and, giving a more general structure than nesting, overlapping structures. This characteristic makes Java's nested class mechanism unnecessary, with the result that the language specification is radically simplified.

- **Ease of code-moving**
  Programmers can easily move code between modules. This is due to a characteristic of difference-based modules: moving code between a super-module and a sub-module does not affect the semantics of the linked modules. As a result, the programmer can perform a kind of refactoring with a high degree of flexibility and without changing the structure of the classes. For example, inter-dependent classes can be split into non-inter-dependent modules without changing the structure of the classes. Ease of code-moving enables smooth shifting from a monolithic prototyping source-code to a modular and extensible source-code.

- **Simplicity**
  Names are inherited only by one mechanism: module inheritance. On the other hand, the specification of Java concerning names is extremely complex. For example, four kinds of classes can be referred to by simple names: (1) Classes belonging to the same package. (2) Classes declared by `import` declarations. (3) Member classes of the outer classes. (4) Member classes of the ancestor classes. The relation between these four mechanisms is far from intuitive.

[1] To be precise, MJ supports a kind of nested class, anonymous classes which are often used for GUI programming in Java.

Chapter 2

MJ Tutorial

2.1 Introduction

This document overviews the run-time environment and the language specifications of MJ. No preliminary knowledge is presumed except the Java language.

All programs presented in this tutorial have already been compiled and included in the jar file `mj.jar` in the MJ distribution package. You can execute these programs if mj is properly installed. Source code of these programs is `lib/Tutorial.java` in the MJ distribution package.

This tutorial excludes an explanation of Collection plug-in and assert2 plug-in. Please refer to Chapter 4 and 5 respectively. Please also refer to Appendix A where the MJ syntax written in BNF is described.

2.2 Preparation

Lets's start by making an empty working directory, where you can run MJ programs. You need to make a directory named `eppout` in the working directory.

```
% mkdir eppout
```

The `eppout` directory is a place to store ".java" files translated by the MJ preprocessor and ".class" files generated by the Java compiler.

2.3 Executing compiled modules : The `mj` command

First of all, we start explanation of how to execute MJ programs before explaining how to write and compile MJ programs.

The file `mj.jar` contains the module `t.m1` whose source code is equivalent to a Java program as in Figure 2.1. (The actual source code written in MJ will be seen later.) Figure 2.1 is a program which defines a class C, and its sub-class SubC.

The module `t.m1` is executable. To execute MJ programs, use the `mj` command, which takes a module name as an argument. The following is the result of the execution.

```
% mj t.m1
----- invoke C#m();
m1:C#m()
----- invoke SubC#m();
m1:C#m()
+ m1:SubC#m()
```

Let's use another module `t.m2` in the `mj.jar`. The `t.m2` is a program which add a difference to the original program, `t.m1`. The source code of `t.m2` would look like the Figure 2.2 if it is written in Java. The program adds 2 lines to the Java source code of `t.m1` .

The following is the result of the execution of the module `t.m2`.

```
% mj t.m2
----- invoke C#m();
m1:C#m()
```

```java
class C  {
  void m() { System.out.println("m1:C#m()"); }
}
class SubC extends C {
  void m() { super.m(); System.out.println("+ m1:SubC#m()"); }
}
class SS {
  public static void main(String[] args){
    System.out.println("----- invoke C#m();");
    C c = new C();
    c.m();
    System.out.println("----- invoke SubC#m();");
    SubC subc = new SubC();
    subc.m();
  }
}
```

Figure 2.1: A Java program which is equivalent to the module `t.m1` .

```java
class C  {
  void m() {
    System.out.println("m1:C#m()");
    System.out.println("+ m2:C#m()");  // Added line.
  }
}
class SubC extends C {
  void m() {
    super.m();
    System.out.println("+ m1:SubC#m()");
    System.out.println("+ m2:SubC#m()");  // Added line.
  }
}
class SS {
  public static void main(String[] args){
    System.out.println("----- invoke C#m();");
    C c = new C();
    c.m();
    System.out.println("----- invoke SubC#m();");
    SubC subc = new SubC();
    subc.m();
  }
}
```

Figure 2.2: A Java program which is equivalent to the module `t.m2` .

```
module t.m1 {
  define class C {
    define C(){}
    define void m(){ System.out.println("m1:C#m()"); }
  }
  define class SubC extends C {
    define SubC(){}
    void m(){ original(); System.out.println("+ m1:SubC#m()"); }
  }
  class SS {
    void main(String[] args){
      System.out.println("----- invoke C#m();");
      C c = new C();
      c.m();
      System.out.println("----- invoke SubC#m();");
      SubC subc = new SubC();
      subc.m();
    }
  }
}
```

Figure 2.3: The source code of the module `t.m1` .

```
+ m2:C#m()
----- invoke SubC#m();
m1:C#m()
+ m2:C#m()
+ m1:SubC#m()
+ m2:SubC#m()
```

MJ has a new feature which allow us to write a difference between the original program and the extended program, as a separate module. Unlike the mechanism of the conventional class inheritance in object-oriented languages, methods in the superclass C themselves can be extended. In the class inheritance mechanism, to extend method functions in a superclass, we must define another subclass. Even if we define the subclass, the methods in the superclass themselves are not extended. MJ enables us to extend the superclass methods without defining another subclass.

2.4 Definition of the modules

2.4.1 The source code of `t.m1`

Figure 2.3 is the source code of `t.m1` written in MJ.

The main difference between MJ programs and Java programs is the `module` declarations, which plays the role of `package` declarations in the Java language. The braces of `module` declarations enclose class definitions.

The second main difference from Java language descriptions is the `define` keyword used within module body. The `define` keyword needs to be specified when classes or methods are defined for the first time. The `define` keyword is not needed for the overriding methods in a subclass. The `define` keyword is not needed for field definitions because fields are never extended. If the `define` keyword is specified to the field definition, it will be ignored.

The `main` method in MJ is not the same as in Java. The class named `SS` plays a special role. `SS` stands for "Startup-Singleton." When a MJ application starts, it generates an instance of the class `SS`, and then its method `main(String[])` is invoked. The `define` keyword is not specified to the class `SS`. The reason will be described in Section 2.10.

In MJ, to invoke the super class's method from subclasses, `original()` is used instead of `super.m()`. In this example, `original()` is invoked in the method `m()` of the class `SubC` which really invokes the method `m()` in the class `C`.

Unlike Java, the MJ compiler never add default constructors to the classes. All constructors should be defined explicitly.

```
module t.m2 extends t.m1 {
  class C {
    void m() { original(); System.out.println("+ m2:C#m()"); }
  }
  class SubC {
    void m() { original(); System.out.println("+ m2:SubC#m()"); }
  }
}
```

Figure 2.4: The source code of the module t.m2 .

```
  module t.m2                      module t.m1
  +-------------+                  +-------------+
  |             |                  |             |
3:|C#m()@t.m2   |      ->        4:|C#m()@t.m1   |
  |             |   difference     |    |        |
  |             |                  |    | inherit |
  |             |                  |    |        |
1:|SubC#m()@t.m2|      ->        2:|SubC#m()@t.m1|
  |             |   difference     |             |
  +-------------+                  +-------------+
```

Figure 2.5: The relationship between the four method fragments.

The keywords `public/protected/private` are not used in MJ. If they are specified , they will be ignored.

2.4.2 The source code of `t.m2`

Figure 2.3 is the source code of `t.m2` written in MJ.

As "`module t.m2 extends t.m1`" at the top indicates, `t.m2` is a module describing just a difference to `t.m1`. `t.m1` is called a *super-module* of `t.m2` and `t.m2` is called a *sub-module* of `t.m1`. The difference between the original program (i.e. the super-module) and the extended program is described in the module body of the sub-module. No `define` is seen in the declaration of class `C` and class `SubC` in the body of the module `t.m2` . These two class declarations are not class definitions. They are the differences between the classes already defined in the super-module and the classes in the extended program. Similarly, no `define` is seen in the method `m()` declarations, which means they are differences between the original methods and the extended methods.

The body of each `m()` method has the expression `original()`, which means to invoke the "original method in the super-module."

The method definition or method differences are sometimes called *method fragments*. The definition of the method fragment `m()` of the class `SubC` in the module `t.m2` is denoted as `SubC#m()@t.m2`.

The relationship between the four method fragments included in `t.m1` or `t.m2` is shown in Figure 2.5. The numbers 1 to 4 in the figure indicates the order in which the method fragments are invoked in the `original()` invocation. First of all, when the method `m()` is invoked, then the first method fragment `SubC#m()@t.m2` will be invoked. If the `original()` is invoked while invoking this method fragment, then the second method fragment `SubC#m()@t.m1` will be invoked. `C#m()@t.m2` and `C#m()@t.m1` will be invoked in the same way.

2.4.3 Automatic loading of the super-modules

To execute the module `t.m2`, just the module name `t.m2` need to be specified as an argument of the `mj` command. The module name `t.m1` does not have to be specified. The `mj` command loads both the modules specified by the arguments and the modules needed to execute them, i.e. all ancestor modules. In this case, when specifying `t.m2`, the `mj` loads `t.m1` also, which is its super-module.

This function is similar to that of the `require` declaration in systems such as Emacs lisp or Common Lisp. In these languages, another package which a given package needs is declared with the `require` declaration. Thus, when loading a package to some system, the required packages are also automatically loaded . In this

```
module t.m3 extends t.m1 {
  class C {
    void m() { original(); System.out.println("+ m3:C#m()"); }
  }
  class SubC {
    void m() { original(); System.out.println("+ m3:SubC#m()"); }
  }
}
```

Figure 2.6: The source code of the module `t.m3` .

```
module t.m4 extends t.m2, t.m3 {
  class C {
    void m() { original(); System.out.println("+ m4:C#m()"); }
  }
  class SubC {
    void m() { original(); System.out.println("+ m4:SubC#m()"); }
  }
}
```

Figure 2.7: The source code of the module `t.m4` .

way, the MJ `extends` declaration has the function of the `require` declaration in Emacs lisp or in Common Lisp.

2.5 Multiple inheritance of modules

In defining modules, more than one super-module can be specified. Suppose first of all, we have the module `t.m3` similar to the module `t.m2` (Figure 2.6).

The result of executing `t.m3` is as follows:

```
% mj t.m3
----- invoke C#m();
m1:C#m()
+ m3:C#m()
----- invoke SubC#m();
m1:C#m()
+ m3:C#m()
+ m1:SubC#m()
+ m3:SubC#m()
```

Thus we can define a `t.m4` module, which inherits both the `t.m2` and the `t.m3` (Figure 2.7).

In this case, both `t.m2` and `t.m3` have `t.m1` as their super-module, so they form a so-called diamond-inheritance. Similar to languages such as CLOS, MJ linearizes modules by topological-sorting.

For example, the `t.m4` execution is as follows. First of all, the linker makes a set consisting of the `t.m4` and its ancestor modules, that is, { `t.m1`, `t.m2`, `t.m3`, `t.m4` }. The linker then, by topological-sorting, linearizes the set so that the inheritance relationship of the modules is preserved. The sorted set is called a linearized list. In this case, the linearized list is (`t.m1 t.m2 t.m3 t.m4`). The difference of each of the modules is added from the beginning of the linearized list and the resulting program is then executed. More concretely, when we denote the resulting program by adding the difference `b` to the program `a` as `a <- b`, we will execute the program `((((e <- t.m1) <- t.m2) <- t.m3) <- t.m4)` where `e` is an empty program. Therefore, the result of the execution is as follows:

```
% mj t.m4
----- invoke C#m();
m1:C#m()
+ m2:C#m()
```

```
+ m3:C#m()
+ m4:C#m()
----- invoke SubC#m();
m1:C#m()
+ m2:C#m()
+ m3:C#m()
+ m4:C#m()
+ m1:SubC#m()
+ m2:SubC#m()
+ m3:SubC#m()
+ m4:SubC#m()
```

The problem remains of how we can decide the order between `t.m2` and `t.m3` which have no inheritance relationship. MJ ignores the order in a super-module declaration (local precedence order), which means nothing is to be changed between the following two module definitions.

```
module t.m4 extends t.m2, t.m3 {...}
module t.m4 extends t.m3, t.m2 {...}
```

The MJ language processor decides the order between `t.m2` and `t.m3` with an algorithm specified by the language specification. (In the current implementation, we adopt a *monotonic* linearization algorithm which uses module names.) There is no way for the programmers to control the order between `t.m2` and `t.m3`. Conversely, the programmers must not write programs with an assumed order between them.

In languages such as CLOS, unlike MJ, priority order is sensitive to the order of superclasses, that is, the local precedence order. It is possible to introduce this function in MJ too, but at present, this function is not introduced.

The worst problem of multiple inheritance in languages such as C++ is name-collisions. This problem has been solved completely in MJ. This is discussed in Section 2.18 .

2.6 The addition of more than one difference : The `mj` command with the `-s` option

The end-user of an application can combine many modules developed independently without writing any lines of code.

For example, although `t.m2` and `t.m3` are completely independent to each other, the end-user may add both of the two differences to `t.m1` at the same time. To do so, using the `-s` option, we can choose modules.

```
% mj -s t.m3 t.m2
----- invoke C#m();
m1:C#m()
+ m2:C#m()
+ m3:C#m()
----- invoke SubC#m();
m1:C#m()
+ m2:C#m()
+ m3:C#m()
+ m1:SubC#m()
+ m2:SubC#m()
+ m3:SubC#m()
```

A set of modules specified as arguments of the `mj` command are called *selected modules*.

When many selected modules are specified as arguments with the `-s` option, how the `mj` command executes them is described below. As above example indicates, suppose that the set of selected modules is { `t.m2`, `t.m3` }. Then, `mj` generates a virtual module named `_bottom`, which extends all selected module elements. This module, which is generated is called the *bottom module*, whose behavior is exactly the same as the module defined as follows:

```
module _bottom extends t.m2, t.m3 {}
```

Then, this is processed as if you executed the command `mj _bottom`. That is, first of all, `mj` makes a set consisting of the module `_bottom` and its ancestor modules. Then, `mj` linearizes the set to construct a linearized

```
module t.hello {
  class SS {
    void main(String[] args){
      original(args);
      System.out.println("Hello.");
    }
  }
}
module t.world extends t.hello {
  class SS {
    void main(String[] args){
      original(args);
      System.out.println("World.");
    }
  }
}
```

Figure 2.8: The source code of the hello world program.

list so that the inheritance relationships are preserved. Then, `mj` adds the differences to the elements of the linearized list from the top in the order of the list and finally execute the resulting application. In this case, the resulting linearized list is: (`t.m1 t.m2 t.m3 _bottom`).

Since the results of execution are not sensitive to the order of the modules in the `extends` declaration, the results of execution are not sensitive to the order of the selected module elements specified in the `-s` option. For example, the following two executions give same results.

```
% mj -s t.m2 t.m3
% mj -s t.m3 t.m2
```

In the current MJ, you cannot add the same difference more than once. Even if you specify the same module names two or more times with the `-s` option, the result is same as if you specified a single module name. For example, the following two commands have a same effect:

```
% mj -s t.m2 t.m3
% mj -s t.m2 -s t.m2 t.m3
```

2.7 Compilation : the `mjc` command

We now will explain how to compile. Figure 2.8 is the source code of a "Hello World" program with two modules.

The two modules may be separated into two files or may be written in a single file, but their file names must all have the `.java` extension. Except for this, unlike Java, there are no restrictions on file names and file paths. It is not permitted to separate a single module into two or more files.

When a single source file has more than one module, the semantics of the program are not sensitive to the module order in the source file.

To compile source files, we use the `mjc` command. Suppose now that each module is written in `Hello.java` and `World.java`. There are the following methods of compiling these two modules:

```
% mjc Hello.java World.java
 or
% mjc World.java Hello.java
 or
% mjc Hello.java; mjc World.java
```

Note that when just `World.java` is specified as follows, `Hello.java` will not be compiled even though `World.java` depends upon it.

```
% mjc World.java
```

The `mjc` process has two phases: one is the EPP preprocessing phase and the other is the javac compilation phase. When the compilation has finished normally, there is an output message as follows:

```
% mjc Hello.java World.java
Preprocessing phase.
Compiling phase.
Done.
```

Almost all compilation-errors will be displayed in the pre-processing phase. But, certain errors may be displayed in the compiling phase.

`mjc` can refer to compiled modules whose class files are in the directories which are included in the CLASSPATH.

Note that the compiled class files are output in the `eppout` directory. For example, if you have executed the `mjc` command at `/a/b/c`, in order to refer to the compiled class files, you must include `/a/b/c/eppout` in the CLASSPATH, not the `/a/b/c` . (Since the `mj` and `mjc` commands automatically include `./eppout` in the CLASSPATH, you do not have to be concerned about the CLASSPATH if you are working on only one directory.)

You can archive all of the class files in the `eppout` directory in the following way. (Here, the `mjclear` command deletes all of the class files in the `eppout`.)

```
% mjclear
% mjc Hello.java
Preprocessing phase.
Compiling phase.
Done.
% cd eppout
% jar cvf0 ../hello.jar .
```

By specifying the jar file to the CLASSPATH, `mjc` or `mj` can refer to them as a library.

```
% setenv CLASSPATH hello.jar:$CLASSPATH
% mjclear
% mjc World.java
Preprocessing phase.
Compiling phase.
Done.
% mj t.world
Hello.
World.
```

2.8 Grouping modules : `mjb` and `mjball` commands

2.8.1 `mjb` command

It is often convenient to combine some of the many small modules from the command line. To do so, you can use the `mjb` command, which is executed in the following way:

```
% mjb bottom m1 m2 m3 ...
```

The `mjb` command generates a module with the name specified as the first argument, which extends all modules specified as the second or later arguments. For example,

```
% mj -s t.m2 t.m3
```

gives the same result as the following execution:

```
% mjb m2m3 t.m2 t.m3
% mj m2m3
```

The execution of the first line is equivalent to the compilation of the following code:

```
module m2m3 extends t.m2, t.m3 {}
```

2.8.2 `mjball` command

Often you want to define a module which extends all modules in the source files. To do this, you can use the `mjball` command, which generates a module, that extends all modules just compiled by the last `mjc` command. For example, suppose you have defined three modules, `a`, `b`, `c`, in `A.java`, `B.java`, `C.java` respectively. In each of the following three ways, the module combining these three modules are executed.

```
% mjc A.java B.java C.java
% mj -s a -s b c
   or
% mjc A.java B.java C.java
% mjb all a b c
% mj all
   or
% mjc A.java B.java C.java
% mjball all
% mj all
```

2.9 Dumping linked applications : the `mjdump` command

The `mjdump` command is needed to execute MJ applications in an environment where `ClassLoader` are not available.

The `mj` command loads classes through a special `ClassLoader` which performs byte-code translation of compiled MJ programs. Therefore, the `mj` command is not available in an environment where the ClassLoader is not available. For example, it is not available in applets.

The `mjdump` command does a byte-code translation like the `mj` command and outputs the results to the file system, but it does not load class files to JavaVM. Here is an example of the execution of the `mjdump` command:

```
% mjdump -s t.m2 t.m3
Dump class : eppout/mjdump/mjc/Start.class
Dump class : eppout/mjdump/mjc/SS.class
Dump class : eppout/mjdump/t/m1/_Delta_java_lang_Object.class
Dump class : eppout/mjdump/t/m1/_Delta_t_m1_C.class
Dump class : eppout/mjdump/t/m2/_Delta_t_m1_C.class
Dump class : eppout/mjdump/t/m3/_Delta_t_m1_C.class
Dump class : eppout/mjdump/t/m1/C.class
Dump class : eppout/mjdump/t/m1/_Delta_t_m1_SubC.class
Dump class : eppout/mjdump/t/m2/_Delta_t_m1_SubC.class
Dump class : eppout/mjdump/t/m3/_Delta_t_m1_SubC.class
Dump class : eppout/mjdump/t/m1/SubC.class
Dump class : eppout/mjdump/mj/lang/ss/_Delta_mjc_SS.class
Dump class : eppout/mjdump/mj/lang/ss/_Delta_mj_lang_ss_SS.class
Dump class : eppout/mjdump/t/m1/_Delta_mj_lang_ss_SS.class
Dump class : eppout/mjdump/mj/lang/ss/SS.class
```

The output is in the `eppout/mjdump` directory. These class files can run in precisely the same way as an ordinary Java application. Since the main method is in the `mjc.Start` class, how to execute it may be as follows:

```
% cd eppout/mjdump
% java mjc.Start
----- invoke C#m();
m1:C#m()
+ m2:C#m()
+ m3:C#m()
----- invoke SubC#m();
m1:C#m()
+ m2:C#m()
+ m3:C#m()
+ m1:SubC#m()
+ m2:SubC#m()
+ m3:SubC#m()
```

```
module mj.lang.ss
{
  define class SS {
    static SS instance;
    define SS(){}
    define void main(String[] args){}
  }
}
```

Figure 2.9: The source code of the class SS.

```
module t.ss.method {
  class SS {
    define int foo(){ return 123; }
    void main(String[] args){
      original(args);
      A a = new A();
      System.out.println(a.bar());  // 1230
    }
  }
  define class A {
    define A(){}
    define int bar(){
      return SS.instance.foo() * 10;
    }
  }
}
```

Figure 2.10: An example of a method invocation of the class `SS`.

Since the output class files with `mjdump` is ordinary Java, and not using the `ClassLoader`, you can use them as applets, MIDlet, etc.

The `mj` command startup slowly (about 1 to 2 seconds), but class files dumped by `mjdump` command can startup as quickly as an ordinary Java application.

2.10 The module `mj.lang.ss` and the class `SS`

2.10.1 The module `mj.lang.ss`

The class `SS` is defined in the module `mj.lang.ss` as in Figure 2.9 .

`t.m1` is referring to the class `SS`. This is to say, `t.m1` is a difference to add to the `mj.lang.ss`, and should to be declared properly as follows:

```
module t.m1 extends mj.lang.ss {...}
```

However, since `mj.lang.ss` is a module to which almost all other modules should extend, the MJ compiler automatically inserts `extends mj.lang.ss` before its compilation.

Currently, the module that is automatically extended by all modules is only the `mj.lang.ss`.

2.10.2 The instance of class `SS`

When the `mj` commend is invoked, it generates an instance of the class `SS`. The instance is assigned to the static field `SS.instance`, then the `main` method of the instance is invoked.

Because current MJ does not fully support static methods, the programmers should use methods of singleton objects. The instance of class `SS` can be used for the purpose. Figure 2.10 is an example of invocation of a method of the class `SS` through the static field `SS.instance` .

```
module t.point.m {
  define class Point {
    int x;
    int y;
    define Point(int x, int y){
      this.x = x; this.y = y;
    }
    define void move(int dx, int dy){ x += dx; y += dy; }
    define int getX(){ return x; }
    define int getY(){ return y; }
  }
  class SS {
    void main(String[] args){
      original(args);
      Point p = new Point(10, 10);
      p.move(1, 2);
      System.out.println(p.getX());  // 11
      System.out.println(p.getY());  // 12
    }
  }
}
```

Figure 2.11: An example of a constructor.

2.11 Constructors

2.11.1 Definitions of constructors

How to define constructors in MJ is slightly different from that of Java, as follows.

- Each class should have at least one definition of constructor in some module. Unlike Java, default constructors are never added by the MJ compiler.
- The definitions of constructors should have the `define` keyword, like method definitions.
- If the head of a constructor body is not one of `super(...)`, `this(...)` or `original(...)`, a statement `super();` is automatically added by the MJ compiler.
- A super constructor invocation with some arguments, such as `super("xxx")`, are currently not supported if the direct super class is a class defined in the Java language. It results in a compile error. If the direct super class is a class defined in the MJ language or the super constructor invocation does not have any arguments, the super constructor invocation will not results in a compile error.

Figure 2.11 is an example of a constructor.

2.11.2 Current implementation of constructors

An invocation of a constructor `new C(...)` will be macro-expanded to `new C()._call_init_C(...)` .

A definition of a constructor `define C(...){...}` will be macro-expanded to the following two method definitions.

```
define C _call_init_C(...){
  _init_C(...);
  return this;
}
define void _init_C(...){...}
```

An definition of an *abstract constructor* `define abstract C(...);` will be macro-expanded to the following two method definitions.

```
module t.abst.m1 {
  define abstract class C {
    define C(){}
    define abstract int foo();
  }
  define class SubC extends C {
    define SubC(){}
    define abstract int bar();
  }
  class SS {
    void main(String[] args){
      original(args);
      SubC c = new SubC();
      System.out.println(c.foo());
      System.out.println(c.bar());
    }
  }
}
module t.abst.m2 extends t.abst.m1 {
  class SubC {
    int foo(){ return 111; }
    int bar(){ return 222; }
  }
}
```

Figure 2.12: Abstract class `C` and non-abstract class `SubC` definitions.

```
define C _call_init_C(...){
  _init_C(...);
  return this;
}
define abstract void _init_C(...);
```

An extension of an constructor `C(...){...}` will be macro-expanded to the following method extension.

```
void _init_C(...){...}
```

A super constructor invocation `super(...)` will be macro-expanded to `_init_S(...)` .
A this constructor invocation `this(...)` will be macro-expanded to `_init_C(...)` .

2.12 Abstract methods

2.12.1 Checking abstract methods at link-time

Classes defined with "`define abstract`" modifiers are called abstract classes. A non-abstract class which has abstract methods will result in an error at link-time. Unlike Java, such a class will not result in an error at compile time because non-abstract methods in another module may override the abstract methods.

Figure 2.12 are examples of abstract class `C` and non-abstract class `SubC` definitions. The respective results of `t.abst.m1` and `t.abst.m2` executions are as follows:

```
% mj t.abst.m1
MJLinker: non-abstract class t.abst.m1.SubC has an abstract method
 : abstract int t.abst.m1::bar()
% mj t.abst.m2
111
222
```

```
module t.color {
  define interface Color {
    int RED = 0xff0000;
    int GREEN = 0x00ff00;
    int BLUE = 0x0000ff;
    define int getRGBCode();
  }
  define class ColorImplementation implements Color {
    define ColorImplementation(){}
    int code = 0x000001;
    int getRGBCode(){ return code; }
  }
  class SS {
    void main(String[] args){
      original(args);
      Color c = new ColorImplementation();
      System.out.println(c.getRGBCode()); // 1
      System.out.println(Color.BLUE);     // 255
    }
  }
}
```

Figure 2.13: An example of a class which implements interface methods.

2.12.2 Interfaces

Interfaces can be used as in Java. The definition of methods in MJ interfaces need `define` modifiers as in MJ classes. When a class implements interfaces, `define` modifiers for method implementations are not required. Figure 2.13 is an example of a class which implements interface methods.

Note that the addition of interface differences is currently not completed to implement. The addition of fields (constants) is implemented but the addition of methods is not yet implemented.

2.12.3 The specification module and the implementation module

Using abstract methods, we can separate a class into a module defining the external interface (called *the specification module*) and into a module defining the implementation (called *the implementation module*). For example, the module in the last section `t.point.m` can be separated into the 3 modules as in Figure 2.14.

The module `t.point` is a specification module defining the external interface of the class `Point`. The module `t.point.implementation` is the implementation module defining the internal implementation of the class `Point`. Note that the module `t.point.test` depends only on the module `t.point`; not on the module `t.point.implementation`.

To execute this program, you need to be aware that specifying only the `t.point.test` as a selected module will result in the following error:

```
% mj t.point.test
MJLinker: non-abstract class t.point.Point has an abstract method
 : abstract int t.point::getY()
```

You should add the implementation module to the selected modules in the following way (or you should combine all modules using the `mjball` command) :

```
% mj -s t.point.implementation t.point.test
11
12
```

2.12.4 Collaboration-Based Modularization

The programmers can define a collaboration that crosscuts more than one class as a separate module, since classes and modules are completely orthogonal in MJ.

```
module t.point {
  define class Point {
    define abstract Point(int x, int y);
    define abstract void move(int dx, int dy);
    define abstract int getX();
    define abstract int getY();
  }
}
module t.point.implementation extends t.point {
  class Point {
    int x;
    int y;
    Point(int x, int y){
      this.x = x; this.y = y;
    }
    void move(int dx, int dy){ x += dx; y += dy; }
    int getX(){ return x; }
    int getY(){ return y; }
  }
}
module t.point.test extends t.point {
  class SS {
    void main(String[] args){
      original(args);
      Point p = new Point(10, 10);
      p.move(1, 2);
      System.out.println(p.getX());  // 11
      System.out.println(p.getY());  // 12
    }
  }
}
```

Figure 2.14: The specification module and the implementation module.

```
  class A {  // class A uses class B
    void m1(B b){ ... b.m3(); ...}
    void m2(){...}
  }
  class B {  // class B uses class A
    void m3(){...}
    void m4(A a){ ... a.m2(); ...}
  }
```

Figure 2.15: Inter-dependent classes containing two collaborations.

```
module A_B {
  define class A {}
  define class B {}
}
module collaboration_m1_m3 extends A_B {
  class A { define void m1(B b){ ... b.m3(); ...} }
  class B { define void m3(){...} }
}
module collaboration_m2_m4 extends A_B {
  class A { define void m2(){...} }
  class B { define void m4(A a){ ... a.m2(); ...} }
}
```

Figure 2.16: Modularized inter-dependent classes.

```
module m1 {
  define class A {}
}
module m2 {
  class SS {
    void main(String[] args){
      original(args);
      A a = new A(); // error
    }
  }
}
```

Figure 2.17: An erroneous program which refers to an invisible name.

Consider the program in Figure 2.15 written in Java. Two classes, classes `A` and `B`, depend on each other; however, these classes actually contain two independent collaborations.

The program can be modularized as in Figure 2.16. The program contains two unrelated modules, `collaboration_m1_m3` and `collaboration_m2_m4`.

Modularization based on collaborations has the following advantages:

- The volume of the source-code on which each module depends decreases. In general, this leads to increased maintainability.
- Because `collaboration_m1_m3` and `collaboration_m2_m4` do not depend on each other, one of the two modules can be compiled and executed even if the other module does not exist. Therefore, these two modules can be developed and tested by different development teams.
- Other variations of application can be provided by means of implementing different versions of collaborations. For example, the module `collaboration_m1_m3` can be replaced by another module `my_collaboration` which contains completely different methods. In this case, existing modules, such as modules `A_B` and `collaboration_m2_m4`, need not be re-compiled.

2.13 Name spaces

2.13.1 Inheritance of name spaces

The inheritance of a module means not only the addition of differences, but also the inheritance of name spaces.

In compiling a module, the compiler regards all modules except super modules as if they do not exist. For example, the program in Figure 2.17 will result in a compilation error. Since `m2` does not declare the `extends m1`, the class A in `m1` is invisible to `m2`.

```
public class A {
  protected static int x = 123;
  public static class B {
    public int getX(){ return x; }
  }
  public static void main(String[] args){
    System.out.println(new B().getX()); // 123
  }
}
```

Figure 2.18: A Java program with nested classes.

```
module t.nest.interface_A_B {
  define class A { }
  define class B {
    define abstract B();
    define abstract int getX();
  }
}
module t.nest.implementation_A_B extends t.nest.interface_A_B {
  class A {
    static int x = 123;
  }
  class B {
    B(){}
    int getX(){ return A.x; }
  }
}
module t.nest.test extends t.nest.interface_A_B {
  class SS {
    void main(String[] args){
      original(args);
      System.out.println(new B().getX());       // 123
    }
  }
}
```

Figure 2.19: An MJ program which represents nested name scopes.

2.13.2 Representation of nested name space

In Java, using packages and nested classes, the programmer can represent a nested name spaces. In MJ, nested name spaces are represented by the inheritance of modules. For example, consider the Java program in Figure 2.18 .

In MJ a program similar to it is as in Figure 2.19 .

The above program will be executed as follows:

```
% mj -s t.nest.implementation_A_B t.nest.test
123
```

The module `interface_A_B` is defining the public names of class `A`, `B` and the module `implementation_A_B` is defining the implementations and the protected names of the class `A` and the class `B`.

In this example, the static field `x` of the class `A` is directly accessed in the class `B` and is written as `A.x`. Thus, all names including field names can be accessed directly from anywhere in the same module or descendant modules. This simple rule realized all functions of the `public/protected` modifiers and nested name spaces in Java and C++.

In Java, protected members are accessible from subclasses even if they are in the different packages, while in MJ, subclassing is not related to the accessibility of names. It is currently not known if this MJ specification

```
module t.javalib.m1
imports java.io.*
imports java.util.Vector
{
}
module t.javalib.m2 extends t.javalib.m1 {
  class SS {
    void main(String[] args){
      original(args);
      File f = new File("f");
      Vector v = new Vector();
    }
  }
}
```

Figure 2.20: An example of the `imports` declarations.

will yield something inconvenient.

2.13.3 Multiple inheritance of name space

The nested class cannot represent more than nested name spaces. MJ enables us to represent more general overlapping name spaces due to the multiple inheritance of modules.

2.14 The `imports` declarations

All Java classes in the CLASSPATH can be used from MJ programs in the same way as in Java. For example, an MJ class can extends a Java class or implements Java interfaces.

Instead of `import` declarations in Java, MJ has `imports` declarations. (Note there is an "`s`" as the last character.) Java classes can be accessed using simple names by declaring `imports` at the top of the module definition.

The effect of the `imports` declaration is inherited by their descendant modules. (It is really convenient.) The `imports` declarations should be declared in modules as low as possible in the module inheritance graph in order to avoid the pollution of name spaces.

Figure 2.20 is a sample program using the `imports` declaration. Note that a "`;`" should not be written at the end of the `imports` declaration.

2.15 `#+comment`

In MJ as in Java, the programmer can use comments such as `//` and `/*...*/`. There is another comment style : `#+`*identifier* `<`*language-construct*`>`. For example, you can comment-out a unit of language construct by writing `#+comment` before it, where the language construct must not have any syntax errors.

The language constructs you can write after `#+`*identifier* are only module definitions, class/interface definitions, member definitions and statements.

`#+`*identifier* can be nested.

To comment-out more than one module at once, you can enclose the modules with brackets: "{" and "}" . The enclosing does not change the semantics of the program. (Note that neither limiting the scope nor grouping the modules occur.)

`#+`*identifier* can be invalidated by writing `//` before it. This is convenient for canceling-out comments temporarily.

Figure 2.21 is a sample program using `#+comment` . The code is equivalent to one as in Figure 2.22 .

The `#+comment<Statement>` is translated to "`;`" (i.e. empty statement). Note that in the `jikes` compiler, writing "`;`" after a return statement etc. will result in the following error: "This statement is unreachable."

The `#+`*identifier* may also be used as a conditional compilation mechanism like the `#ifdef` in C language.

```
mjc -J-Ddebug=true
```

```
module t.sharpPlus.m1 {
  class SS {
    void main(String[] args){
      original(args);
 #+comment
      System.out.println("aaa");
      //#+comment
      {
        System.out.println("bbb");
      }
      System.out.println("ccc");
    }
  }
}
 #+comment
{
  module t.sharpPlus.m1{
 #+comment
    class C { }
  }
 #+comment
  module t.sharpPlus.m2{
  }
}
```

Figure 2.21: An example of a program using `#+comment` .

```
module t.sharpPlus.m1 {
  class SS {
    void main(String[] args){
      original(args);
      {
        System.out.println("bbb");
      }
      System.out.println("ccc");
    }
  }
}
```

Figure 2.22: A program equivalent to Figure 2.21 .

```
module t.uses.a uses t.uses.b {
  define class A {
    define A(){}
    define B getB(){ return new B(); }
  }
}
module t.uses.b uses t.uses.a {
  define class B {
    define B(){}
    define A getA(){ return new A(); }
  }
}
```

Figure 2.23: An example of `uses` declarations.

The above option in the compiler means to validate language constructs after the `#+debug` and invalidate language constructs after the `#-debug`.

2.16 The uses declarations

The `extends` relation between modules is not allowed to be cyclic. However, the first stage of program development often involves inter-dependency among the modules. The `uses` declarations are introduced for these situations. Like `extends`, a `uses` declaration inherits name spaces, but is allowed to have a cyclic relation. Figure 2.23 is an example of `uses` declarations.

The `uses` declaration restricts separate compilations. For example, suppose that the two modules in Figure 2.23 are written respectively in the `A.java` and the `B.java` files. To compile this program, you must give both the `A.java` and `B.java` files to the `mjc` arguments, otherwise you will get compilation errors, or will do a compile with reference to old compilation results.

The `extends` relation gives constraints to the module linearization, whereas `uses` gives no constraints.

There is no difference between `extends` and `uses` except for these rules.

When a module `m2` extends names that are defined in a module `m1`, `m2` must be lower than `m1` in the linearized list. Otherwise the program would not be right. Therefore, in this case, the programmers must use "`m2 extends m1`" instead of "`m2 uses m1`" . In other cases, that is, when the programmers simply use names defined in `m1` within `m2`, they can use "`m2 uses m1`" instead of "`m2 extends m1` .

2.17 Dynamic loading of classes

MJ supports dynamic loading of classes as well as Java.

Figure 2.24 is an example of dynamic loading.

In current implementation of MJ, the constructor of the loaded class is not invoked automatically. Please do not forget to invoke the method `_init_A()` explicitly if necessary.

2.18 The Fully-Qualified-Name (FQN)

The name-collision problem that is incurred by multiple inheritance is fully resolved in MJ. In Java, the name-collision problem caused by `import` declarations is resolved by the idea of *fully-qualified-names* (FQNs) of classes. In MJ, this idea is applied to all names including field and method names in order to resolve the problem.

In MJ, each name has a unique FQN. Each FQN consists of "the module name which first defined the name" and "a simple name". If a simple name is used at one point in the source-code and more than one candidate which has the same simple name is accessible at that point, the compiler will report an error because the reference is ambiguous. Two names defined at different places are never regarded as identical by the compiler. A name definition never shadows another name. If an error is reported because of an ambiguous reference to a name, the programmer can always avoid this error by using the FQN of the name instead of the simple name.

We assume that the uniqueness of the module names is guaranteed by other mechanisms or rules, such as the naming convention adding as a prefix the domain name of the vendor, as in Java.

```
module t.forName {
  define class A {
    static int x = 123;
  }
  class SS {
    void main(String[] args){
      original(args);
      try {
        Class c = Class.forName("t.forName.A");
        A a = (A)c.newInstance();
        System.out.println(a.x); // 123
      } catch (Throwable e){
        e.printStackTrace(System.err);
        System.err.println(e.getMessage());
        throw new Error();
      }
    }
  }
}
```

Figure 2.24: An example of dynamic class loading.

In MJ, an FQN which consists “the defining module name `m`” and “the simple name `n`” is expressed as “`FQN[m::n]`”. This is illustrated in the program in Figure 2.25 .

In this program, when the programmer invokes the method ambiguously such as in `a.m()`, instead of using FQN, the compiler will report the following error:

```
% mjc Tutorial.java
Preprocessing phase.
Tutorial.java:322: MJ: Reference to m() of t.fqn.m1.A is ambiguous. :
    t.fqn.m2::m
    t.fqn.m3::m

      System.out.println(a.m()); // error
                           ^
1 errors
```

2.19 The implementation defect and the complementary modules

In MJ, when combining many modules, a phenomenon called *implementation defect* may occur. The *complementary module* is a module which complements this defect.

The following is a detailed explanation of the implementation defect using the program in Figure 2.26 . The module `t.compl.orig` defines an abstract class `S` and its subclass `A`. The module `t.comp.sub` defines a new subclass `B`. On the other hand, the module `t.comp.abst` defines an abstract method `m` in the class `S` and implements it in the subclass `A`. Both `t.comp.sub` and `t.compl.abst` are complete programs without any errors at link-time. However, when trying to use both of the modules simultaneously, we will get the following error at link-time:

```
% mj -s t.compl.sub t.compl.abst
MJLinker: non-abstract class t.compl.sub.B has an abstract method
 : abstract int t.compl.abst::m()
```

This is to say, we will get an error because no one implements the method `m` in the class `B`. The phenomenon, which we call an implementation defect is that which occurs when we combine two modules each of which runs properly, and un-implemented abstract methods may be yielded as in the above case.

It is generally impossible to complement the defect automatically. Someone must implement the complementary modules, which complement the defect after understanding the specifications of the two modules.

```
module t.fqn.m1 {
  define class A {
    define A(){}
  }
}
module t.fqn.m2 extends t.fqn.m1 {
  class A {
    define int m(){ return 2; }
  }
}
module t.fqn.m3 extends t.fqn.m1 {
  class A {
    define int m(){ return 3; }
  }
}
module t.fqn.m4 extends t.fqn.m2, t.fqn.m3 {
  class A {
    int FQN[t.fqn.m2::m](){ return original() * 10; }
    int FQN[t.fqn.m3::m](){ return original() * 100; }
  }
  class SS {
    void main(String[] args){
      original(args);
      A a = new A();
      //System.out.println(a.m()); // error
      System.out.println(a.FQN[t.fqn.m2::m]());  // 20
      System.out.println(a.FQN[t.fqn.m3::m]());  // 300
    }
  }
}
```

Figure 2.25: An example of FQN.

```
module t.compl.orig {
  define abstract class S { }
  define class A extends S { }
}
module t.compl.sub extends t.compl.orig {
  define class B extends S { }
}
module t.compl.abst extends t.compl.orig {
  class S {
    define abstract int m();
  }
  class A {
    int m(){ return 1; }
  }
}
```

Figure 2.26: An example of the implementation defect between two modules.

```
module t.compl.compl_sub_abst complements t.compl.sub, t.compl.abst {
  class B {
    int m(){ return 2; }
  }
}
```

Figure 2.27: The complementary module.

MJ intends to support that end-users without any detailed knowledge of implementations can configure applications they would like to construct by combining modules. The MJ linker has the function of automatic linking complementary modules for convenience of the end-users. The complementary module complementing the `t.comp.sub` and the `t.compl.abst` is defined as a module having the `complements` declaration, such as "`complements t.comp.sub, t.compl.abst`".

For example, suppose that the complementary module in Figure 2.27 was implemented and was put in some directory visible through the CLASSPATH. The compiler processes the "`complements`" declaration in the same way as an "`extends`" declaration, except that the compiler adds information of module names to be complemented to the compiled binary.

Next, when end-users try to combine `t.comp.sub` and `t.compl.abst` as follows, the linker will automatically find the complementary module `t.compl.compl_sub_abst` in the CLASSPATH and link it together.

```
% mj -s t.compl.sub t.compl.abst
```

The end-users need not be aware of the implementation defect problem because of the automatic complementation mechanism of the MJ linker.

Chapter 3

The execution environment

3.1 Supported environment

The shell-scripts for compilation and execution are now being tested in the following environment.

- Linux, IBM JDK1.1.8
- Windows2000, cygwin 1.3.2, Sun JDK1.2.2
- Windows98, cygwin 1.3.5, Sun JDK1.3.1_01

Furthermore, the dumped class have been tested not only in the above environment but also in the following execution environment.

- Sun MIDP Emulator 1.0
- i-mode Java F503i, P503i, So503i (Japanese cellular phones of NTT DoCoMo)

3.2 How to install

Download the latest version from the MJ Home Page. By unzipping the downloaded zip file, you will find the directory "`mj`". Add "`mj/bin`" to your PATH. The CLASSPATH need not to be set.

3.3 Description of each shell-script

3.3.1 `mjc [options] File.java ...`

The `mjc` command compiles source files specified as arguments: and outputs class files under the `eppout` directory.

The `mjc` command outputs a compiling log to `MJ/MJC.log` .

Unlike `javac`, `mjc` does not compile files which depend on specified files. All of the source files you wish to compile must be specified as arguments.

The `mjc` command writes all the module names contained in the compiled source files to the `MJ/MJall.log` that is used by the `mjball` command.

The `mjc` command internally invokes the EPP (pre-processor written in Java) and the java compiler.

Options:

- `-J`*xxx* passes *xxx* to a `java` command option.
- `-E`*xxx* passes *xxx* to an `epp` command option.
- `-h` gives the indication of short option descriptions.

An arbitrary number of `-J`*xxx* and `-E`*xxx* may be specified as options. For example:

```
% mjc -J-mx512m -J-Ddebug=true -E-plug-in -Ejp.go.etl.epp.typeof *.java
```

3.3.2 `mj [options] module [args...]`

The `mj` command links and loads modules specified by arguments (called "*selected modules*") to JavaVM and then executes them. The `mj` command automatically links ancestor modules of the selected modules and the necessary complementary modules in CLASSPATH.

The `mj` command does not require the `eppout` directory under the current directory.

Options:

- `-J`*xxx* passes *xxx* to the `java` command option.
- `-E`*xxx* passes *xxx* to the `epp` command option.
- `-s` *m* adds *m* to the set of selected modules.
- `-log` outputs a log file to the `MJ/MJ.log` .
- `-h` gives the indication of short option descriptions.

An arbitrary number of `-J`*xxx*, `-E`*xxx* and `-s m` may be specified as options.

`[args...]` are passed to the "`void main(String[] args)`" of the application.

The order of selected modules has no effect on the running.

3.3.3 `mjb bottom m1 m2 ...`

The `mjb` command generates a module with a name specified by the first arguments (called the "*bottom module*"), which extends all modules specified in the second or later arguments; and then outputs the class files of the bottom module under the `eppout` directory.

The bottom module that is generated is available in the same way as a compiled module from a source file or modules in CLASSPATH. For example:

```
% mjb m1m2 m1 m2
% mjb m1m2m3 m1m2 m3
% mj m1m2m3
...
```

Unlike the `mj` command, the `mjb` command does not add complementary modules for the specified modules.

3.3.4 `mjball all`

The `mjball` command generates a bottom module, which extends to all modules included in *.java files compiled by the execution of the last `mjc` command; and then outputs it under the `eppout` directory. For example:

```
% mjc T1.java
% mjball all
% mj all
```

This command refers to the contents of the `MJ/MJall.log` file generated by the `mjc` command.

3.3.5 `mjdump [options] module`

The `mjdump` command links all the selected modules and then outputs the class files as an ordinary Java application under the `eppout/mjdump` directory. Like the `mj` command, the `mjdump` command automatically links ancestor modules of the selected modules and the necessary complementary modules in CLASSPATH.

The dumped application can be executed by the "`java mjc.Start [args...]`" command, as usual Java applications. For example:

```
% mjdump all
...
% cd eppout/mjdump
% java mjc.Start
```

The `mjdump` command accepts the same options as in the `mj` command.

```
java.lang.Error: test!
        at m3._Delta_ss_SS.m1_testA(_Delta_ss_SS.java:114)
        at m1._Delta_ss_SS.main(_Delta_ss_SS.java:42)
        at m2._Delta_ss_SS.main(_Delta_ss_SS.java:76)
        at m3._Delta_ss_SS.main(_Delta_ss_SS.java:106)
        at merge._Delta_ss_SS.main(_Delta_ss_SS.java:168)
        at anonymousClass._Delta_ss_SS.main(_Delta_ss_SS.java:205)
        at very.longlong.module.name._Delta_ss_SS.main(_Delta_ss_SS.java:249)
        at very.longlong.another.module.name._Delta_ss_SS.main(_Delta_ss_SS.java:262)
        at mjc.MJLinker.startMain(MJLinker.java:186)
        at mjc.MJLinker.doLink(MJLinker.java:100)
        at mjc.MJC_Epp_globalProcessAfterTypeCheckingPass.call(PlugIn.java:123)
        at jp.go.etl.epp.epp.FileInfo.globalProcessAfterTypeCheckingPass(FileInfo.java:175)
        at jp.go.etl.epp.epp.Epp.globalEpp(Epp.java:470)
        at jp.go.etl.epp.epp.Epp.processFiles(Epp.java:339)
        at jp.go.etl.epp.epp.Epp.processFilesAndCatchEppUserError(Epp.java:325)
        at jp.go.etl.epp.epp.Epp.eppMain(Epp.java:55)
        at jp.go.etl.epp.epp.Epp.main(Epp.java:22)
test!
java.lang.Error: Error during executing main.
```

Figure 3.1: An example of stack trace.

3.3.6 `mjclear`

The `mjclear` command removes all *.java and *.class safely under the `eppout` directory.

```
% mjclear
Checking files under "eppout"...
Start removing files under eppout...
Done.
No Java files were translated.
%
```

By reading the top of the files, this command checks whether *.java files have been generated by EPP. When non-generated *.java files or files other than *.java or *.class exist under `eppout`, the `mjclear` command removes none of files at all and stops with a message output.

```
% mjclear
Checking files under "eppout"...

The followings are not generated files.
eppout/memo.txt

No files were removed. Please remove them manually.
```

3.4 How to read stack trace

Figure 3.1 is an example of stack trace.

When JavaVM outputs `(Compiled Code)` instead of line numbers, turn JIT off to get the line numbers in the stack trace output. For example:

```
% ( setenv MJJAVA 'java -Djava.compiler=' ; mj all )
```

Only the part above “`mjc.MJLinker.startMain(MJLinker.java:186)`” is related to your application. Ignore the lines below it.

Figure 3.2 describe how to read each line.

```
        at m3._Delta_ss_SS.m1_testA(_Delta_ss_SS.java:114)
           -- *1
                     ----- *2
                           -------- *3
           --                               -------------- *4
```

*1 indicates invoking a method in the module `m3`
*2 indicates invoking a method in the class `ss.SS`
*3 indicates invoking a method named `m1::testA`
*4 indicates processing at line 114 in the class `ss.SS` in the module `m3`

Figure 3.2: How to read each line of the stack trace.

```
End initializing MJClassInfo of merge0.C at all .
MJClassInfo:
class merge0.C extends java.lang.Object {
  int merge1::x
  int merge2::x
  public final native void java::wait(long) throws java.lang.InterruptedExceptio
n
  public final void java::wait(long, int) throws java.lang.InterruptedException
  public final void java::wait() throws java.lang.InterruptedException
  protected void java::finalize() throws java.lang.Throwable
  public native int java::hashCode()
  public final native void java::notifyAll()
  public final native void java::notify()
  public final native java.lang.Class java::getClass()
  int merge1::foo()
  int merge2::foo()
  protected native java.lang.Object java::clone() throws java.lang.CloneNotSuppo
rtedException
  public boolean java::equals(java.lang.Object)
  public java.lang.String java::toString()
  }
```

Figure 3.3: An example of MJClassInfo in the log file.

3.5 How to read log files

3.5.1 How and where to output a log.

The `mjc` command outputs a log to the file `MJ/MJC.log` .

The `mj` and `mjdump` commands do not output logs by default. If the `-log` option is specified, it outputs a log to the file `MJ/MJ.log` .

The log content described below is common to `MJC.log` and `MJ.log` files.

3.5.2 MJClassInfo

When getting an error in type-checking, you may want to know "what methods the class C has from the view point of the module `m`." In this case, you should locate the output from the MJClassInfo command in the log file.

In order to do this, search the log file with the string "`End initializing MJClassInfo of C at m`". For example, Figure 3.3 is the MJClassInfo of the class `merge0.C` in view from the module `all` .

(You see the modifiers of `public`, `protected`, however, they are ignored in MJ.)

```
Start initializing MJClassInfo of m1.SubA at all .
Start initializing MJClassInfo of m1.A at all .
        m1.A#m1::a at m2
                 overrides m1.A#m1::a at m1 .
        m1.A#m1::a at m3
                 overrides m1.A#m1::a at m2 .
End initializing MJClassInfo of m1.A at all .
MJClassInfo:
<The output of MJClassInfo of mj.A is omitted.>
        m1.SubA#m1::a at m1
                 overrides m1.A#m1::a at m3 .
        m1.SubA#m1::a at m2
                 overrides m1.SubA#m1::a at m1 .
        m1.SubA#m2::m2a at m2
                 overrides m1.A#m2::m2a at m2 .
        m1.SubA#m1::a at m3
                 overrides m1.SubA#m1::a at m2 .
        m1.SubA#m3::m3a at m3
                 overrides m1.A#m3::m3a at m3 .
End initializing MJClassInfo of m1.SubA at all .
```

Figure 3.4: An example of log of processing method overriding.

3.5.3 Override

You may want to know “which method fragment overrides which method fragment from the view point of the module `m`.” See the processing of override logs in making MJClassInfo.

For example, Figure 3.4 is the log of processing methods of the `mj.SubA` class viewed from the `all` module.

From this, it is possible to see that the `m1::a` method is overridden in the following order.

```
m1.A#m1::a at m1
m1.A#m1::a at m2
m1.A#m1::a at m3
m1.SubA#m1::a at m1
m1.SubA#m1::a at m2
m1.SubA#m1::a at m3
```

3.5.4 LinearizedUsingModules

You can see how the modules are linearized viewed from a certain module. Search the log file with the string: `linearizedUsingModules for ModuleName`. For example, a module list viewed from `m2` is:

```
linearizedUsingModules for m2 :
  mj.lang.ss
  m1
  m2
```

Chapter 4

Collection plug-in

4.1 Introduction

Collection plug-in provides built-in parameterized types and some useful language constructs. The characteristics of Collection plug-in are summarized as follows:

- Statically type-checked vector, hashtable and iterator.
- Convenient language constructs : foreach and ifNull .
- Automatic wrap/unwrap of int type.
- Simple specification and implementation.
- High composability with other language extension plug-ins.

4.2 Basic rules

Collection types (Vec<T>, Table<Key,Value>, Iter<T> and NullOr<T>) can be used as built-in type names with type parameters.

Only the int type and the reference types (class, interface, T[] or Collection types) are permitted for type parameters.

When specifying int as a type-parameter, arguments or return values for Collection types are automatically wrapped/unwrapped to Integer, if necessary.

When nesting the Collection types, the space character should be inserted between two ">", otherwise, you would get a syntax error because the compiler could not parse the syntax.

4.3 Vec<T>

Vec<T> is a vector, an array with a variable length. The assignment/reference to the element for a specific index can be executed at $O(1)$. Adding/deleting the last element is allowed, like stacks. Using the method toArray(), Vec<T> can be translated to a T[]. (Unlike the Java 2 collection libraries, the type is not Object[] .)

Figure 4.1 is a program using Vec<T> .

"{}" can be used as a initialization expression of a variable declaration for Vec<T>. "{}" has the same effect as "new Vec<T>()". Initial elements can be listed in "{}".

For the instances of Vec<T> the foreach statement (explained later) is carefully implemented so that it becomes as efficient as possible. It will loop almost as fast as the for loop for T[] . (I have not timed this yet, however.) Moreover, the execution of the foreach statement for Vec<T> does not consume any heap memory.

4.4 NullOr<T> and ifNull statement

NullOr<T> is a built-in type that denotes explicitly "null or T". NullOr<T> forces "null value check" to programmers.

The type NullOr<T> is used as follows:

```
Vec<String> vec = {};  // Initialize vec to new Vec<String>() .
vec.push("aaa");
vec.push("bbb");
vec.push("ccc");
System.out.println(vec.get(2));  // -> ccc
foreach (String s in vec){
  System.out.println(s);         // -> aaa, bbb, ccc
}
vec.put(2, "xxx");  // Update the second element of vec.
foreach (int index, String s in vec){
  System.out.println(index+ ":"+ s); // -> 0:aaa, 1:bbb, 2:xxx
}
System.out.println(vec.size());  // -> 3
System.out.println(vec.pop());   // -> xxx
System.out.println(vec.size());  // -> 2
```

Figure 4.1: An example of a program using `Vec<T>` .

```
void test(){
  String s = find("a") ifNull { s = "default"; };
  ...
}
NullOr<String> find(String key){
  if (...) {
    ...
    return str;
  } else {
    return null;
  }
}
```

Values of the type `NullOr<T>` do not accept any operations of the `T`. Values of the type `NullOr<T>` must be converted to `T` using `ifNull` statement.

Values of the type `NullOr<T>` can be used as ordinary first class data; in other words, they can be assigned to another variables, passed to arguments for method invocations or used as return values.

Values of type `NullOr<S>`, `S`, `null` can be assigned to the variables of `NullOr<T>`, where `S` is a subtype of `T` (or `T` itself).

There are two forms of `ifNull` statement. One is a local variable declaration followed by `ifNull` and the other is variable assignment followed by `ifNull`. They are macro-expanded as the following:

```
        { T var = exp ifNull block; rest; }
->
        { T var;
          T tmp = exp;
          if (tmp != null)
            var = tmp;
          else
            block
          rest;
        }

        var = exp ifNull block;
->
        { T tmp = exp;
          if (tmp != null)
            var = tmp;
          else
            block
        }
```

```
Table<String,int> table = {};  // Initialize vec to new Table<Key,Value>() .
table.put("aaa", 111);
table.put("bbb", 222);
table.put("ccc", 333);
int x1 = table.get("aaa")
      ifNull { throw new Error(); };
System.out.println(x1);  // -> 111
foreach (int val in table.valueIter()){
  System.out.println(val);            // -> 111, 222, 333
}
table.put("aaa", 999);  // Update the element corresponding to "aaa" .
foreach (String key, int val in table){
  System.out.println(key+ ":"+ val);  // -> aaa:999, bbb:222, ccc:333
}
```

Figure 4.2: An example of a program using `Table<key,Value>` .

In the former form, the scope of the declared variable ranges within the block of `ifNull` and over statements subsequent to the `ifNull` statement. If the declared variable is not assigned in the block following `ifNull`, a reference to the variable causes a compile error because the variable is not definitely assigned. For example,

```
NullOr<int> x = 123;
int y = x ifNull { /* do nothing */ };
System.out.println(y);
```

the above program will result in the following error.

```
Test.java:19: Variable y may not have been initialized.
      (System.out).println(y);
                           ^
1 error
```

4.5 Table<Key,Value>

An instance of a `Table<Key,Value>` is a hashtable. Figure 4.2 is an example using the `Table<key,Value>` .

"{}" can be used as an initialization expression for `Table<Key,Value>`. "{}" has the same effect as "`new Table<Key,Value>()`". The current implementation does not support specifying initial elements within "{}" .

Values are added with the method `put(key,value)`, that will throw a `NullPointerException` if the value is `null`.

Values are referred to with the method `get(key)`, that returns a value of `NullOr<Value>` . The programmer should convert the type to the type `Value` before doing some operations on the value. This is a safer design than `Map` of the Java 2 collection library.

4.6 Iter<T>

An instance of `Iter<T>` is an iterator. `Iter<T>` is a type of returned values of `vec.iter()`, `table.keyIter()` or `table.valueIter()`. Currently, there is no other way to instantiate `Iter<T>`.

`Iter<T>` has the following two methods. The usage is similar to Iterator in the Java 2 collection library. Of course, an explicit cast is not needed.

```
boolean hasNext()
T next()
```

The collection value should not be updated while its iterator is in use. The behavior in doing so is not defined by the specifications.

```
found:
  foreach (Vec<String> vec in vecOfVec){
    foreach (String s in vec){
      if (s.equals(keyString)) break found;
    }
  }
```

Figure 4.3: An example of a break with a label.

4.7 foreach statement

4.7.1 One variable foreach

One variable foreach is the following form:

```
foreach (Value val in exp) { ... }
```

The type of exp must be either of int, T[], Vec<T> or Iter<T>.

- If exp is int, the val runs from 0 to exp-1 in order.
- If exp is T[], the val runs from exp[0] to exp[exp.length-1] in order.
- If exp is Vec<T>, the val runs from exp.get(0) to exp.get(exp.size()-1) in order.
- If exp is Iter<T>, the val runs over exp.next() values until exp.hasNext() becomes false.

The exp is evaluated only once before entering the loop.

4.7.2 Two variables foreach

Two variable foreach is the following form:

```
foreach (Key key, Value val in exp) { ... }
```

The type of exp must be either T[], Vec<T> or Table<Key,Value> .

- If exp is T[], the key runs from 0 to exp.length-1,
  and the val runs from exp[0] to exp[exp.length-1] in order.
- If exp is Vec<T>, the key runs from 0 to exp.size()-1,
  and the val runs from exp.get(0) to exp.get(exp.size()-1) in order.
- If exp is Table<Key,Value>, the pair (key, val) runs over the keys and their corresponding values in the hashtable.

The exp is evaluated only once before entering the loop.

4.7.3 break and continue

The break and continue statements inside foreach behave as expected. The break with a label also behaves as expected. Figure 4.3 is an example of a break with a label.

In the current implementation, the continue with a label does not behave as expected. (javac gives a compile time error.) This is a bug to be fixed in the future.

4.7.4 Updating elements

While looping with foreach, elements of the data structure can be updated only in the following cases.

- While looping for T[], any element may be assigned.
- While looping Vec, any element may be assigned using vec.put(i,x). (push and pop are not allowed.)

With these exceptions , updating the collection elements during foreach loop is not allowed. The behavior in doing so is not defined by the specification.

```
Table<String, Vec<String> > table = {};
Vec<String> vec1 = {"aaa", "bbb", "\n"};
Vec<String> vec2 = {"\"\t\""};
table.put("key1", vec1);
table.put("key2", vec2);
Vec<String> v = table.print("table:").get("key1")
  ifNull{ throw new Error(); };
```

Figure 4.4: An example of program using `print(String)` methods.

```
table:
{
 {
  "key2",
  {
    "\"\t\"",
  }
 }, {
  "key1",
  {
    "aaa",
    "bbb",
    "\n",
  }
 }
}
```

Figure 4.5: The output of Figure 4.4 .

4.8 The wrap/unwrap of `int`

The `int` can be specified as a type-parameter. In its implementation, the value of `int` is wrapped in an `Integer` that is invisible to programmers. Since Integers for x in the range $10 \leq x < 1024$ is cached, no memory is allocated from the heap when wrapping. The range for caching is customizable with the following method:

```
jp.go.etl.epp.util.Wrapper.setCacheRange(min, max);
```

The wrap/unwrap of primitive types other than `int` is currently not implemented.

4.9 The `print` method

`Vec` and `Table` have the method `print(String)` that is convenient for debugging.

- Their nested structure can be printed with indentations.
- Because the print method invocation returns the object itself as a value, "`.print(str)`" can be embedded in an internal point of expression.
- If the element is a String, outputs can be attached with the quotation marks and escape sequences.

Figure 4.4 is an example of program using `print(String)` methods. Figure 4.5 is the output of the program.

The printing format for a user defined class is customizable. For details, examine the source code of the module `t.collection.printFunction` in the file `lib/Tutorial.java`, included in the distribution package of MJ.

4.10 Future work

4.10.1 Integration to "generics"

The feature of "generics" will be introduced in the future version of Java, however, it lacks useful features, such as `foreach` statement, which are already provided by Collection plug-in. Collection plug-in should be re-implemented as an extension of the "generics." Because we have already implemented a prototype version of GJ plug-in, it will not be difficult task.

4.10.2 The >> problem

Like `g++`, when parameterized classes are nested, you must insert a space between two "`>`" as `Vec<Vec<T> >`, but not as `Vec<Vec<T>>`. This problem must be remedied.

4.11 Collection plug-in Reference Manual

```
Syntax
------
Type:
        <original alternatives>
        Vec < Type >
        Table < Type , Type >
        Iter < Type >
        NullOr < Type >
Statement:
        <original alternatives>
        foreach ( Type Identifier in Expression ) Block
                # The type of Expression is int, T[], Vec<T> or Iter<T> .

        foreach ( Type Identifier , Type Identifier in Expression ) Block
                # The type of Expression is T[], Vec<T> or Table<Key,Value> .

        Identifier = Expression ifNull Block ;
        Type Identifier = Expression ifNull Block ;

Constructors and methods
------------------------
Constructors of Vec<T>:
        Vec<T>()
        Vec<T>(Object[])
        Vec<T>(Vector)

Methods of Vec<T>:
        void push(T)
        T pop()
        T top()
        void put(int, T)
        T get(int)
        int size()
        T[] toArray()
        Iter<T> iter()
        Vec<T> print(String)

Constructors of Table<Key,Value>
        Table<Key,Value>()

Methods of Table<Key,Value>:
        void put(Key, Value)    // NOTE: unable to put null.
        NullOr<Value> get(Key)
        void remove(Key)
        boolean containsKey(Key)
        Iter<Key> keyIter()
        Iter<Value> valueIter()
        Table<Key,Value> print(String)

Methods of Iter<T>:
        boolean hasNext()
        T next()
```

Idioms of Vec

Making newVec by applying the function f to each element of vec.

```
Vec<T> newVec = {};
foreach (T x in oldVec) { newVec.push(f(x)); }
```

Adding vec2 to the tail of vec1.

```
foreach (T x in vec2) { vec1.push(x); }
```

Searching the first element satisfying the predicate p from vec.

```
T val = null;
foreach (T x in vec) {
  if (p(x)) {
    val = x;
    break;
  }
}
if (val == null) { ... } else { ... }
```

Substituting each element x of vec to f(x) if it satisfies the predicate p.

```
foreach (int index, T x in vec) {
  if (p(x)) { vec.put(index, f(x)); }
}
```

Idioms of Table

Default values.

```
Value val = table.get(key) ifNull { val = defaultValue; };
```

Registering default value to the table.

```
Value val = table.get(key) ifNull {
  val = defaultValue;
  table.put(key, val); // Do not forget.
};
```

In case the entry should exist.

```
Value val = table.get(key) ifNull { throw new Error("fatal"); };
```

Processing something only when the entry exist.

```
Value val = table.get(key) ifNull { return; };
...
```

Processing something only when the entry does not exist.

```
if (! table.containsKey(key)) { ... }
```

In case the entry may not exist.

```
Value val = table.get(key) ifNull { val = null; };
if (val == null) { ... } else { ... }
```

Looping over all key.

```
foreach (Key x in table.keyIter()) { ... }
```

Idioms of Table<Key,Vec<Value> >

Adding val to vec corresponding to the key.

```
Vec<Value> vec = table.get(key)
  ifNull {
    vec = new Vec<Value>();
    table.put(key, vec); // Do not forget.
  };
vec.push(val);
```

Chapter 5

assert2 plug-in

The assert2 plug-in emulates `assert` statement introduced by Java2 1.4 .

The emulated `assert` statement throws `java.lang.Error` if the specified conditional expression is false.

There are two styles of `assert` statements.

```
Statement:
        assert Expression;
        assert Expression : Expression;
```

The `assert` statement with one expression will report more detailed message than Java2 1.4, if the specified expression is an comparison operator. For example,

```
double d = 10;
assert(d / 2 > d + 2);
```

will produce the following error message.

```
java.lang.Error: Assertion failed : (5.0 > 12.0) is not true.
```

The `assert` statement with two expressions is the same as in Java2 1.4 . For example,

```
int y = 0;
assert y != 0 : "Zero divide.";
```

will produce the following error message.

```
java.lang.Error: Assertion failed : Zero divide.
```

If the user set the value of property `NDEBUG` to true, all `assert` statements will be macro-expanded to empty statements. For example:

```
% mjc -J-DNDEBUG=true Tutorial.java
Preprocessing phase.
Compiling phase.
Done.
% mj t.assert
assert test.
```

Chapter 6

How to rewrite applications written in Java to MJ

It is better not to think about writing a modular program from the beginning. The following process is recommended:

1. We assume that the Java source code to be rewritten is sufficiently refactored and its function is sufficiently stable. Otherwise, we shall be puzzled whether bugs are due to the original application, the rewriting process or the MJ language preprocessor.

2. Write a unit test program to test all the functions of the application. This test program should be used for every iterations of rewriting of the source code.

3. By reading the source code, find code not allowed in MJ such as inner classes and super constructor invocation for Java classes, then rewrite them so that they will be accepted by MJ.

4. If the application size is not too large, gather all classes in a single file.

5. Rewrite the Java source code to MJ. This is the work with the highest risk.

   First, enclose the whole program with a single module block. Attach `define` to the classes and methods definitions. Rewrite `super` invocations to `original` invocations. Leave `public/protected/private` unchanged because `mjc` ignores them.

   During rewriting, the class model of the application should be unchanged. The images after linking should be completely equivalent to the original Java code, to avoid enbugging.

   Classes obviously not inter-dependent to others may be separated to other modules.

   If we pass the compiler and the test, we have succeeded.

6. Finally, pick up "crosscutting concerns" to other modules. It is better not to make small modules unless it is necessary.

Chapter 7

Bugs and Limitations

7.1 About error messages

EPP may report three kinds of error messages.

- Error messages that EPP reports. These are reported when the program is illegal as ordinary Java language.
- Error messages that the `mjc` plug-in reports. These are reported when the program is illegal as MJ language. In most cases an "`MJ:`" will be displayed at the top of the message. (However, it may NOT be attached to some error messages.)
- Error messages that `javac` reports. These are reported when the program is illegal as ordinary Java language. (The MJ compiler should check all errors properly at the preprocessing phase, however, in the current implementation, some errors are not checked during this phase.)

If EPP or the `mjc` plug-in reports a `FATAL ERROR`, it is a bug of the `mjc` command. The `mjc` command should not report a `FATAL ERROR` even if the input program is erroneous.

Error messages, which are not easy to understand, are also bugs of the `mjc` command. (When you get an unreasonable error, try `mjc` with the option `-E-nocatch`. Sometimes more reasonable errors are reported.)

7.2 Known bugs and how to avoid them

7.2.1 Initializing a field with a field value causes a compilation-error.

Example:

```
module m {
 define class C {
  int x = 0;
  int y = x;  // error
 }
}
```

How to avoid this: Use FQN.

```
module m {
 define class C {
  int x = 0;
  int y = FQN[m::x];
 }
}
```

7.2.2 Within an anonymous class, fields and methods of an outer class are not accessible .

How to avoid this: After assigning `this` value to a final local variable `outer` in the following way, access to field/method with `outer.foo` or `outer.foo()`.

```
final Draw outer = this; // Declaration of the final variable "outer".
b.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent e){
    Graphics g = outer.getGraphics();  // Invoking outer class's method.
    ...;
  }});
```

7.2.3 "MJC: FATAL ERROR: findMethodInfo:..." is reported.

You get a FATAL ERROR when you define a method with the same signature as one defined in the Java libraries.

Example:

```
module m1 {
  define class A {
    define boolean equals(Object x) { return false; }
  }
}
```

Error messages:

```
Preprocessing phase.
        Test16.java:9: findMethodInfo: More than one MethodInfo found.
          java::equals
          m1::equals

            define boolean equals(Object x) { return false; }
            ^
        java.lang.Error: MJC: FATAL ERROR : findMethodInfo: More than one MethodInfo found.
          java::equals
          m1::equals
```

How to avoid this : Remove the define as it is not needed when overriding the methods defined in the Java libraries.

7.2.4 Specifying MJ classes in an argument of a constructor of an anonymous class causes a type-error.

How to avoid this: Cast the type of Java class explicitly needed in the constructor arguments.

7.2.5 Adding methods to an MJ interface causes a compilation-error.

How to avoid this : None. Discontinue adding a difference to interfaces.

7.2.6 Invoking a protected method from another class does not cause a compilation error.

For example, when invoking `protected Object clone()` from the other classes, the compiler cannot detect the error, and a ClassCastException is reported at run-time.

How to avoid this : In the case of a `clone`, you should override it as follows:

```
define class C {
  Object clone() throws CloneNotSupportedException {
    return original();
  }
}
```

7.2.7 "MJ:Type dependency loop is detected" is reported .

This error message is reported when referring to a class whose declaration part has an error.

How to avoid this : Correct the error in the class declaration part.

7.2.8 A compilation error is caused when an MJ class implements a certain kind of Java Interface.

Example:

```
b.java:2: MJ: In b.B at module b : Definition-method overrides another method. : equals
  define class B implements java.security.Principal {
  ^
1 errors
```

How to avoid this : Define a Java class which implements the interface using Java language, and make the compiled class file which is accessible from the CLASSPATH. Then, define a MJ class which extends the class.

7.2.9 Referring to constants (static final fields) with simple names causes a compile-error during the compilation phase of mjc.

How to avoid this : Instead of using simple names, refer to constants with the syntax `ClassName.CONSTANT`.

7.2.10 Using `C.class` causes a `FATAL ERROR`.

How to avoid this : Replace it with `Class.forName("C")` .

Chapter 8

Acknowledgements

In MJ prototype version, we used "javassist" by Shigeru Chiba at Tokyo Institute of Technology as a byte-code translation tool. In the current version, we are using JavaClass (BCEL) by Markus Dahm.

We thank AIST visiting researcher, Akira TANAKA for great co-operations as an alpha tester.

We also thank Takaho MATSUZAKI at Knowledge Information Research Institute Co. for various testing and creating demonstration applets.

This research is supported by JST (PRESTO, Japan Science and Technology Corporation) and AIST (National Institute of Advanced Industrial Science and Technology).

Appendix A

Syntax

We describe here only the differences in syntax between MJ and Java.

- "`*`" means the preceding non-terminal repeats more than zero times.
- "`<e>`" means an empty string.
- "`<original alternatives>`" means there are other alternatives defined by The Java language specification.
- Capitalized words mean non-terminals and non-capitalized words mean terminals if there are no special notes.

```
MJCompilationUnit:
        MJModule*

MJModule:
        ;
        { MJModule* }
        #+ SharpPlusExpression MJModule
        #- SharpPlusExpression MJModule
        module MJModuleName MJModuleHeaderDeclaration* {
            MJTypeDeclaration*}

MJModuleHeaderDeclaration:
        complements MJModuleNameList
        extends MJModuleNameList
        uses MJModuleNameList
        imports MJImportedPackageNameList

MJModuleNameList:
        MJModuleName
        MJModuleName , MJModuleNameList

MJImportedPackageNameList:
        MJImportedPackageName
        MJImportedPackageName , MJImportedPackageNameList

MJImportedPackageName:
        Name
        Name . *

MJTypeDeclaration:
        MJClass
        ;
        #+ SharpPlusExpression MJTypeDeclaration
        #- SharpPlusExpression MJTypeDeclaration
```

```
MJClass:
        MJDefine Modifier* MJClassKeyword MJName Extends Implements {
                MJClassBodyDeclaration* }

MJClassKeyword:
        class
        interface

MJClassBodyDeclaration:
        MJFieldDeclaration
        MJConstructorDeclaration
        MJMethodDeclaration
        ;
        #+ SharpPlusExpression MJClassBodyDeclaration
        #- SharpPlusExpression MJClassBodyDeclaration

MJFieldDeclaration:
        MJDefine Modifier* Type VariableDecorators   // FQN is not allowed.

MJConstructorDeclaration:
        MJDefine Modifier* MJName ( FormalParameterList )
                Throws MJMethodBody

MJMethodDeclaration:
        MJDefine Modifier* Type MJName ( FormalParameterList )
                Throws MJMethodBody

MJDefine:
        <e>
        define

MJMethodBody:
        ;
        Block

Statement:
        <original alternatives>
        #+ SharpPlusExpression Statement
        #- SharpPlusExpression Statement

Primary:
        <original alternatives>
        original ArgumentList
        MJFQN                                   // this field access
        MJFQN ArgumentList                      // this method invocation
        Expression . MJFQN                      // field access
        Expression . MJFQN ArgumentList         // method invocation
        Name . MJFQN                            // static field access
        Name . MJFQN ArgumentList               // static method invocation
        MJFQN . MJFQN                           // static field access
        MJFQN . MJFQN ArgumentList              // static method invocation
        new MJFQN ...                           // instance creation

Type:
        <original alternatives>
        MJFQN

MJName:
        Identifier
        MJFQN

MJFQN:
        FQN [ MJModuleName :: Identifier ]
        // This "FQN" is not a non-terminal symbol, but a token.
```

```
MJModuleName:
        Name

SharpPlusExpression:
        Identifier
        String                  // e.g. "debug=true"

VariableDecorators:
        <original alternatives>

ArgumentList:
        <original alternatives>

Expression:
        <original alternatives>

Name:
        <original alternatives>

Modifier:
        <original alternatives>

Identifier:
        <original alternatives>

String:
        <original alternatives>
```