Getting Started with Javassist

Shigeru Chiba

Next page

1. Reading and writing bytecode

Javassist is a class library for dealing with Java bytecode. Java bytecode is stored in a binary file called a class file. Each class file contains one Java class or interface.

The class `Javassist.CtClass` is an absatract representation of a class file. A `CtClass` (compile-time class) object is a handle for dealing with a class file. The following program is a very simple example:

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("test.Rectangle");
cc.setSuperclass(pool.get("test.Point"));
cc.writeFile();
```

This program first obtains a `ClassPool` object, which controls bytecode modification with Javassist. The `ClassPool` object is a container of `CtClass` object representing a class file. It reads a class file on demand for constructing a `CtClass` object and records the constructed object for responding later accesses. To modify the definition of a class, the users must first obtain from a `ClassPool` object a reference to a `CtClass` object representing that class. `get()` in `ClassPool` is used for this purpose. In the case of the program shown above, the `CtClass` object representing a class `test.Rectangle` is obtained from the `ClassPool` object and it is assigned to a variable `cc`. The `ClassPool` object returned by `getDfault()` searches the default system search path.

From the implementation viewpoint, `ClassPool` is a hash table of `CtClass` objects, which uses the class names as keys. `get()` in `ClassPool` searches this hash table to find a `CtClass` object associated with the specified key. If such a `CtClass` object is not found, `get()` reads a class file to construct a new `CtClass` object, which is recorded in the hash table and then returned as the resulting value of `get()`.

The `CtClass` object obtained from a `ClassPool` object can be modified (details of how to modify a `CtClass` will be presented later). In the example above, it is modified so that the superclass of `test.Rectangle` is changed into a class `test.Point`. This change is reflected on the original class file when `writeFile()` in `CtClass()` is finally called.

`writeFile()` translates the `CtClass` object into a class file and writes it on a local disk. Javassist also provides a method for directly obtaining the modified bytecode. To obtain the bytecode, call `toBytecode()`:

```
byte[] b = cc.toBytecode();
```

You can directly load the `CtClass` as well:

```
Class clazz = cc.toClass();
```

`toClass()` requests the context class loader for the current thread to load the class file represented by the `CtClass`. It returns a `java.lang.Class` object representing the loaded class. For more details, please see this section below.

Defining a new class

To define a new class from scratch, `makeClass()` must be called on a `ClassPool`.

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.makeClass("Point");
```

This program defines a class `Point` including no members. Member methods of `Point` can be created with factory methods declared in `CtNewMethod` and appended to `Point` with `addMethod()` in `CtClass`.

`makeClass()` cannot create a new interface; `makeInterface()` in `ClassPool` can do. Member methods in an interface can be created with `abstractMethod()` in `CtNewMethod`. Note that an interface method is an abstract method.

Frozen classes

If a `CtClass` object is converted into a class file by `writeFile()`, `toClass()`, or `toBytecode()`, Javassist freezes that `CtClass` object. Further modifications of that `CtClass` object are not permitted. This is for warning the developers when they attempt to modify a class file that has been already loaded since the JVM does not allow reloading a class.

A frozen `CtClass` can be defrost so that modifications of the class definition will be permitted. For example,

```
CtClasss cc = ...;
    :
cc.writeFile();
cc.defrost();
cc.setSuperclass(...);    // OK since the class is not frozen.
```

After `defrost()` is called, the `CtClass` object can be modified again.

If `ClassPool.doPruning` is set to `true`, then Javassist prunes the data structure contained in a `CtClass` object when Javassist freezes that object. To reduce memory consumption, pruning discards unnecessary attributes (`attribute_info` structures) in that object. For example, `Code_attribute` structures (method bodies) are discarded. Thus, after a `CtClass` object is pruned, the bytecode of a method is not accessible except method names, signatures, and annotations. The pruned `CtClass` object cannot be defrost again. The default value of `ClassPool.doPruning` is `false`.

To disallow pruning a particular `CtClass`, `stopPruning()` must be called on that object in advance:

```
CtClasss cc = ...;
cc.stopPruning(true);
    :
cc.writeFile();                              // convert to a class file.
// cc is not pruned.
```

The `CtClass` object `cc` is not pruned. Thus it can be defrost after `writeFile()` is called.

> **Note:** While debugging, you might want to temporarily stop pruning and freezing and write a modified class file to a disk drive. `debugWriteFile()` is a convenient method for that purpose. It stops pruning, writes a class file, defrosts it, and turns pruning on again (if it was initially on).

Class search path

The default `ClassPool` returned by a static method `ClassPool.getDefault()` searches the same path that the underlying JVM (Java virtual machine) has. *If a program is running on a web application server such as JBoss and*

Tomcat, the `ClassPool` object may not be able to find user classes since such a web application server uses multiple class loaders as well as the system class loader. In that case, an additional class path must be registered to the `ClassPool`. Suppose that `pool` refers to a `ClassPool` object:

```
pool.insertClassPath(new ClassClassPath(this.getClass()));
```

This statement registers the class path that was used for loading the class of the object that `this` refers to. You can use any `Class` object as an argument instead of `this.getClass()`. The class path used for loading the class represented by that `Class` object is registered.

You can register a directory name as the class search path. For example, the following code adds a directory `/usr/local/javalib` to the search path:

```
ClassPool pool = ClassPool.getDefault();
pool.insertClassPath("/usr/local/javalib");
```

The search path that the users can add is not only a directory but also a URL:

```
ClassPool pool = ClassPool.getDefault();
ClassPath cp = new URLClassPath("www.javassist.org", 80, "/java/", "org.javassist.");
pool.insertClassPath(cp);
```

This program adds "http://www.javassist.org:80/java/" to the class search path. This URL is used only for searching classes belonging to a package `org.javassist`. For example, to load a class `org.javassist.test.Main`, its class file will be obtained from:

```
http://www.javassist.org:80/java/org/javassist/test/Main.class
```

Furthermore, you can directly give a byte array to a `ClassPool` object and construct a `CtClass` object from that array. To do this, use `ByteArrayClassPath`. For example,

```
ClassPool cp = ClassPool.getDefault();
byte[] b = a byte array;
String name = class name;
cp.insertClassPath(new ByteArrayClassPath(name, b));
CtClass cc = cp.get(name);
```

The obtained `CtClass` object represents a class defined by the class file specified by `b`. The `ClassPool` reads a class file from the given `ByteArrayClassPath` if `get()` is called and the class name given to `get()` is equal to one specified by `name`.

If you do not know the fully-qualified name of the class, then you can use `makeClass()` in `ClassPool`:

```
ClassPool cp = ClassPool.getDefault();
InputStream ins = an input stream for reading a class file;
CtClass cc = cp.makeClass(ins);
```

`makeClass()` returns the `CtClass` object constructed from the given input stream. You can use `makeClass()` for eagerly feeding class files to the `ClassPool` object. This might improve performance if the search path includes a large jar file. Since a `ClassPool` object reads a class file on demand, it might repeatedly search the whole jar file for every class file. `makeClass()` can be used for optimizing this search. The `CtClass` constructed by `makeClass()` is kept in the `ClassPool` object and the class file is never read again.

The users can extend the class search path. They can define a new class implementing `ClassPath` interface and give an instance of that class to `insertClassPath()` in `ClassPool`. This allows a non-standard resource to be included in the search path.

---

2. ClassPool

A `ClassPool` object is a container of `CtClass` objects. Once a `CtClass` object is created, it is recorded in a `ClassPool` for ever. This is because a compiler may need to access the `CtClass` object later when it compiles source code that refers to the class represented by that `CtClass`.

For example, suppose that a new method `getter()` is added to a `CtClass` object representing `Point` class. Later, the program attempts to compile source code including a method call to `getter()` in `Point` and use the compiled code as the body of a method, which will be added to another class `Line`. If the `CtClass` object representing `Point` is lost, the compiler cannot compile the method call to `getter()`. Note that the original class definition does not include `getter()`. Therefore, to correctly compile such a method call, the `ClassPool` must contain all the instances of `CtClass` all the time of program execution.

Avoid out of memory

This specification of `ClassPool` may cause huge memory consumption if the number of `CtClass` objects becomes amazingly large (this rarely happens since Javassist tries to reduce memory consumption in various ways). To avoid this problem, you can explicitly remove an unnecessary `CtClass` object from the `ClassPool`. If you call `detach()` on a `CtClass` object, then that `CtClass` object is removed from the `ClassPool`. For example,

```
CtClass cc = ... ;
cc.writeFile();
cc.detach();
```

You must not call any method on that `CtClass` object after `detach()` is called. However, you can call `get()` on `ClassPool` to make a new instance of `CtClass` representing the same class. If you call `get()`, the `ClassPool` reads a class file again and newly creates a `CtClass` object, which is returned by `get()`.

Another idea is to occasionally replace a `ClassPool` with a new one and discard the old one. If an old `ClassPool` is garbage collected, the `CtClass` objects included in that `ClassPool` are also garbage collected. To create a new instance of `ClassPool`, execute the following code snippet:

```
ClassPool cp = new ClassPool(true);
// if needed, append an extra search path by appendClassPath()
```

This creates a `ClassPool` object that behaves as the default `ClassPool` returned by `ClassPool.getDefault()` does. Note that `ClassPool.getDefault()` is a singleton factory method provided for convenience. It creates a `ClassPool` object in the same way shown above although it keeps a single instance of `ClassPool` and reuses it. A `ClassPool` object returned by `getDefault()` does not have a special role. `getDefault()` is a convenience method.

Note that `new ClassPool(true)` is a convenient constructor, which constructs a `ClassPool` object and appends the system search path to it. Calling that constructor is equivalent to the following code:

```
ClassPool cp = new ClassPool();
cp.appendSystemPath();  // or append another path by appendClassPath()
```

Cascaded ClassPools

If a program is running on a web application server, creating multiple instances of `ClassPool` might be necessary; an instance of `ClassPool` should be created for each class loader (i.e. container). The program should create a `ClassPool` object by not calling `getDefault()` but a constructor of `ClassPool`.

Multiple `ClassPool` objects can be cascaded like `java.lang.ClassLoader`. For example,

```
ClassPool parent = ClassPool.getDefault();
ClassPool child = new ClassPool(parent);
child.insertClassPath("./classes");
```

If `child.get()` is called, the child `ClassPool` first delegates to the parent `ClassPool`. If the parent `ClassPool` fails to find a class file, then the child `ClassPool` attempts to find a class file under the `./classes` directory.

If `child.childFirstLookup` is true, the child `ClassPool` attempts to find a class file before delegating to the parent `ClassPool`. For example,

```
ClassPool parent = ClassPool.getDefault();
ClassPool child = new ClassPool(parent);
child.appendSystemPath();          // the same class path as the default one.
child.childFirstLookup = true;     // changes the behavior of the child.
```

Changing a class name for defining a new class

A new class can be defined as a copy of an existing class. The program below does that:

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
cc.setName("Pair");
```

This program first obtains the `CtClass` object for class `Point`. Then it calls `setName()` to give a new name `Pair` to that `CtClass` object. After this call, all occurrences of the class name in the class definition represented by that `CtClass` object are changed from `Point` to `Pair`. The other part of the class definition does not change.

Note that `setName()` in `CtClass` changes a record in the `ClassPool` object. From the implementation viewpoint, a `ClassPool` object is a hash table of `CtClass` objects. `setName()` changes the key associated to the `CtClass` object in the hash table. The key is changed from the original class name to the new class name.

Therefore, if `get("Point")` is later called on the `ClassPool` object again, then it never returns the `CtClass` object that the variable `cc` refers to. The `ClassPool` object reads a class file `Point.class` again and it constructs a new `CtClass` object for class `Point`. This is because the `CtClass` object associated with the name `Point` does not exist any more. See the followings:

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
CtClass cc1 = pool.get("Point");   // cc1 is identical to cc.
cc.setName("Pair");
CtClass cc2 = pool.get("Pair");    // cc2 is identical to cc.
CtClass cc3 = pool.get("Point");   // cc3 is not identical to cc.
```

`cc1` and `cc2` refer to the same instance of `CtClass` that `cc` does whereas `cc3` does not. Note that, after `cc.setName("Pair")` is executed, the `CtClass` object that `cc` and `cc1` refer to represents the `Pair` class.

The `ClassPool` object is used to maintain one-to-one mapping between classes and `CtClass` objects. Javassist never allows two distinct `CtClass` objects to represent the same class unless two independent `ClassPool` are created. This is a significant feature for consistent program transformation.

To create another copy of the default instance of `ClassPool`, which is returned by `ClassPool.getDefault()`, execute the following code snippet (this code was already shown above):

```
ClassPool cp = new ClassPool(true);
```

If you have two `ClassPool` objects, then you can obtain, from each `ClassPool`, a distinct `CtClass` object representing the same class file. You can differently modify these `CtClass` objects to generate different versions of the class.

Renaming a frozen class for defining a new class

Once a `CtClass` object is converted into a class file by `writeFile()` or `toBytecode()`, Javassist rejects further modifications of that `CtClass` object. Hence, after the `CtClass` object representing `Point` class is converted into a class file, you cannot define `Pair` class as a copy of `Point` since executing `setName()` on `Point` is rejected. The following code snippet is wrong:

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
cc.writeFile();
cc.setName("Pair");    // wrong since writeFile() has been called.
```

To avoid this restriction, you should call `getAndRename()` in `ClassPool`. For example,

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
cc.writeFile();
CtClass cc2 = pool.getAndRename("Point", "Pair");
```

If `getAndRename()` is called, the `ClassPool` first reads `Point.class` for creating a new `CtClass` object representing `Point` class. However, it renames that `CtClass` object from `Point` to `Pair` before it records that `CtClass` object in a hash table. Thus `getAndRename()` can be executed after `writeFile()` or `toBytecode()` is called on the the `CtClass` object representing `Point` class.

3. Class loader

If what classes must be modified is known in advance, the easiest way for modifying the classes is as follows:

- 1. Get a `CtClass` object by calling `ClassPool.get()`,
- 2. Modify it, and
- 3. Call `writeFile()` or `toBytecode()` on that `CtClass` object to obtain a modified class file.

If whether a class is modified or not is determined at load time, the users must make Javassist collaborate with a class loader. Javassist can be used with a class loader so that bytecode can be modified at load time. The users of Javassist can define their own version of class loader but they can also use a class loader provided by Javassist.

3.1 The `toClass` method in `CtClass`

The `CtClass` provides a convenience method `toClass()`, which requests the context class loader for the current thread to load the class represented by the `CtClass` object. To call this method, the caller must have appropriate permission; otherwise, a `SecurityException` may be thrown.

The following program shows how to use `toClass()`:

```
public class Hello {
    public void say() {
        System.out.println("Hello");
    }
}

public class Test {
    public static void main(String[] args) throws Exception {
        ClassPool cp = ClassPool.getDefault();
        CtClass cc = cp.get("Hello");
        CtMethod m = cc.getDeclaredMethod("say");
        m.insertBefore("{ System.out.println(\"Hello.say():\"); }");
        Class c = cc.toClass();
        Hello h = (Hello)c.newInstance();
        h.say();
    }
}
```

`Test.main()` inserts a call to `println()` in the method body of `say()` in `Hello`. Then it constructs an instance of the modified `Hello` class and calls `say()` on that instance.

Note that the program above depends on the fact that the `Hello` class is never loaded before `toClass()` is invoked. If not, the JVM would load the original `Hello` class before `toClass()` requests to load the modified `Hello` class. Hence loading the modified `Hello` class would be failed (`LinkageError` is thrown). For example, if `main()` in `Test` is something like this:

```
public static void main(String[] args) throws Exception {
    Hello orig = new Hello();
    ClassPool cp = ClassPool.getDefault();
    CtClass cc = cp.get("Hello");
        :
}
```

then the original `Hello` class is loaded at the first line of `main` and the call to `toClass()` throws an exception since the class loader cannot load two different versions of the `Hello` class at the same time.

If the program is running on some application server such as JBoss and Tomcat, the context class loader used by `toClass()` might be inappropriate. In this case, you would see an unexpected `ClassCastException`. To avoid this exception, you must explicitly give an appropriate class loader to `toClass()`. For example, if `bean` is your session bean object, then the following code:

```
CtClass cc = ...;
Class c = cc.toClass(bean.getClass().getClassLoader());
```

would work. You should give `toClass()` the class loader that has loaded your program (in the above example, the class of the `bean` object).

`toClass()` is provided for convenience. If you need more complex functionality, you should write your own class loader.

3.2 Class loading in Java

In Java, multiple class loaders can coexist and each class loader creates its own name space. Different class loaders can load different class files with the same class name. The loaded two classes are regarded as different ones. This feature enables us to run multiple application programs on a single JVM even if these programs include different classes with the same name.

> **Note:** The JVM does not allow dynamically reloading a class. Once a class loader loads a class, it cannot reload a modified version of that class during runtime. Thus, you cannot alter the definition of a class after the JVM loads it. However, the JPDA (Java Platform Debugger Architecture) provides limited ability for reloading a class. See Section 3.6.

If the same class file is loaded by two distinct class loaders, the JVM makes two distinct classes with the same name and definition. The two classes are regarded as different ones. Since the two classes are not identical, an instance of one class is not assignable to a variable of the other class. The cast operation between the two classes fails and throws a *`ClassCastException`*.

For example, the following code snippet throws an exception:

```
MyClassLoader myLoader = new MyClassLoader();
Class clazz = myLoader.loadClass("Box");
Object obj = clazz.newInstance();
Box b = (Box)obj;    // this always throws ClassCastException.
```

The `Box` class is loaded by two class loaders. Suppose that a class loader CL loads a class including this code snippet. Since this code snippet refers to `MyClassLoader`, `Class`, `Object`, and `Box`, CL also loads these classes (unless it delegates to another class loader). Hence the type of the variable `b` is the `Box` class loaded by CL. On the other hand, `myLoader` also loads the `Box` class. The object `obj` is an instance of the `Box` class loaded by `myLoader`. Therefore, the last statement always throws a `ClassCastException` since the class of `obj` is a different verison of the `Box` class from one used as the type of the variable `b`.

Multiple class loaders form a tree structure. Each class loader except the bootstrap loader has a parent class loader, which has normally loaded the class of that child class loader. Since the request to load a class can be delegated along this hierarchy of class loaders, a class may be loaded by a class loader that you do not request the class loading. Therefore, the class loader that has been requested to load a class C may be different from the loader that actually loads the class C. For distinction, we call the former loader *the initiator of C* and we call the latter loader *the real loader of C*.

Furthermore, if a class loader CL requested to load a class C (the initiator of C) delegates to the parent class loader PL, then the class loader CL is never requested to load any classes referred to in the definition of the class C. CL is not the initiator of those classes. Instead, the parent class loader PL becomes their initiators and it is requested to load them. *The classes that the definition of a class C referes to are loaded by the real loader of C.*

To understand this behavior, let's consider the following example.

```
public class Point {    // loaded by PL
    private int x, y;
    public int getX() { return x; }
        :
}

public class Box {      // the initiator is L but the real loader is PL
    private Point upperLeft, size;
    public int getBaseX() { return upperLeft.x; }
        :
}

public class Window {    // loaded by a class loader L
    private Box box;
    public int getBaseX() { return box.getBaseX(); }
}
```

Suppose that a class `Window` is loaded by a class loader L. Both the initiator and the real loader of `Window` are L. Since the definition of `Window` refers to `Box`, the JVM will request L to load `Box`. Here, suppose that L delegates this task to the parent class loader PL. The initiator of `Box` is L but the real loader is PL. In this case, the initiator of `Point` is not L but PL since it is the same as the real loader of `Box`. Thus L is never requested to load `Point`.

Next, let's consider a slightly modified example.

```
public class Point {
    private int x, y;
    public int getX() { return x; }
        :
}

public class Box {      // the initiator is L but the real loader is PL
    private Point upperLeft, size;
    public Point getSize() { return size; }
        :
}

public class Window {    // loaded by a class loader L
    private Box box;
    public boolean widthIs(int w) {
        Point p = box.getSize();
        return w == p.getX();
    }
}
```

Now, the definition of `Window` also refers to `Point`. In this case, the class loader L must also delegate to PL if it is requested to load `Point`. *You must avoid having two class loaders doubly load the same class.* One of the two loaders must delegate to the other.

If L does not delegate to PL when `Point` is loaded, `widthIs()` would throw a ClassCastException. Since the real loader of `Box` is PL, `Point` referred to in `Box` is also loaded by PL. Therefore, the resulting value of `getSize()` is an instance of `Point` loaded by PL whereas the type of the variable `p` in `widthIs()` is `Point` loaded by L. The JVM regards them as distinct types and thus it throws an exception because of type mismatch.

This behavior is somewhat inconvenient but necessary. If the following statement:

```
Point p = box.getSize();
```

did not throw an exception, then the programmer of `Window` could break the encapsulation of `Point` objects. For example, the field `x` is private in `Point` loaded by PL. However, the `Window` class could directly access the value of `x` if L loads `Point` with the following definition:

```
public class Point {
    public int x, y;    // not private
    public int getX() { return x; }
        :
}
```

For more details of class loaders in Java, the following paper would be helpful:

> Sheng Liang and Gilad Bracha, "Dynamic Class Loading in the Java Virtual Machine",
> *ACM OOPSLA'98*, pp.36-44, 1998.

3.3 Using `javassist.Loader`

Javassist provides a class loader `javassist.Loader`. This class loader uses a `javassist.ClassPool` object for reading a class file.

For example, `javassist.Loader` can be used for loading a particular class modified with Javassist.

```
import javassist.*;
import test.Rectangle;

public class Main {
  public static void main(String[] args) throws Throwable {
     ClassPool pool = ClassPool.getDefault();
     Loader cl = new Loader(pool);

     CtClass ct = pool.get("test.Rectangle");
     ct.setSuperclass(pool.get("test.Point"));

     Class c = cl.loadClass("test.Rectangle");
     Object rect = c.newInstance();
         :
  }
}
```

This program modifies a class `test.Rectangle`. The superclass of `test.Rectangle` is set to a `test.Point` class. Then this program loads the modified class, and creates a new instance of the `test.Rectangle` class.

If the users want to modify a class on demand when it is loaded, the users can add an event listener to a `javassist.Loader`. The added event listener is notified when the class loader loads a class. The event-listener class must implement the following interface:

```
public interface Translator {
    public void start(ClassPool pool)
```

```
            throws NotFoundException, CannotCompileException;
        public void onLoad(ClassPool pool, String classname)
            throws NotFoundException, CannotCompileException;
    }
```

The method `start()` is called when this event listener is added to a `javassist.Loader` object by `addTranslator()` in `javassist.Loader`. The method `onLoad()` is called before `javassist.Loader` loads a class. `onLoad()` can modify the definition of the loaded class.

For example, the following event listener changes all classes to public classes just before they are loaded.

```
    public class MyTranslator implements Translator {
        void start(ClassPool pool)
            throws NotFoundException, CannotCompileException {}
        void onLoad(ClassPool pool, String classname)
            throws NotFoundException, CannotCompileException
        {
            CtClass cc = pool.get(classname);
            cc.setModifiers(Modifier.PUBLIC);
        }
    }
```

Note that `onLoad()` does not have to call `toBytecode()` or `writeFile()` since `javassist.Loader` calls these methods to obtain a class file.

To run an application class `MyApp` with a `MyTranslator` object, write a main class as following:

```
    import javassist.*;

    public class Main2 {
      public static void main(String[] args) throws Throwable {
         Translator t = new MyTranslator();
         ClassPool pool = ClassPool.getDefault();
         Loader cl = new Loader();
         cl.addTranslator(pool, t);
         cl.run("MyApp", args);
      }
    }
```

To run this program, do:

```
    % java Main2 arg1 arg2...
```

The class `MyApp` and the other application classes are translated by `MyTranslator`.

Note that *application* classes like `MyApp` cannot access the *loader* classes such as `Main2`, `MyTranslator`, and `ClassPool` because they are loaded by different loaders. The application classes are loaded by `javassist.Loader` whereas the loader classes such as `Main2` are by the default Java class loader.

`javassist.Loader` searches for classes in a different order from `java.lang.ClassLoader`. `ClassLoader` first delegates the loading operations to the parent class loader and then attempts to load the classes only if the parent class loader cannot find them. On the other hand, `javassist.Loader` attempts to load the classes before delegating to the parent class loader. It delegates only if:

- the classes are not found by calling `get()` on a `ClassPool` object, or
- the classes have been specified by using `delegateLoadingOf()` to be loaded by the parent class loader.

This search order allows loading modified classes by Javassist. However, it delegates to the parent class loader if it fails to find modified classes for some reason. Once a class is loaded by the parent class loader, the other classes referred to in that class will be also loaded by the parent class loader and thus they are never modified. Recall that all the classes referred to in a class C are loaded by the real loader of C. *If your program fails to load a modified class,* you should make sure whether all the classes using that class have been loaded by `javassist.Loader`.

3.4 Writing a class loader

A simple class loader using Javassist is as follows:

```
import javassist.*;

public class SampleLoader extends ClassLoader {
    /* Call MyApp.main().
     */
    public static void main(String[] args) throws Throwable {
        SampleLoader s = new SampleLoader();
        Class c = s.loadClass("MyApp");
        c.getDeclaredMethod("main", new Class[] { String[].class })
         .invoke(null, new Object[] { args });
    }

    private ClassPool pool;

    public SampleLoader() throws NotFoundException {
        pool = new ClassPool();
        pool.insertClassPath("./class"); // MyApp.class must be there.
    }

    /* Finds a specified class.
     * The bytecode for that class can be modified.
     */
    protected Class findClass(String name) throws ClassNotFoundException {
        try {
            CtClass cc = pool.get(name);
            // modify the CtClass object here
            byte[] b = cc.toBytecode();
            return defineClass(name, b, 0, b.length);
        } catch (NotFoundException e) {
            throw new ClassNotFoundException();
        } catch (IOException e) {
            throw new ClassNotFoundException();
        } catch (CannotCompileException e) {
            throw new ClassNotFoundException();
        }
    }
}
```

The class `MyApp` is an application program. To execute this program, first put the class file under the `./class` directory, which must *not* be included in the class search path. Otherwise, `MyApp.class` would be loaded by the default system class loader, which is the parent loader of `SampleLoader`. The directory name `./class` is specified by `insertClassPath()` in the constructor. You can choose a different name instead of `./class` if you want. Then do as follows:

```
% java SampleLoader
```

The class loader loads the class `MyApp` (`./class/MyApp.class`) and calls `MyApp.main()` with the command line parameters.

This is the simplest way of using Javassist. However, if you write a more complex class loader, you may need detailed knowledge of Java's class loading mechanism. For example, the program above puts the `MyApp` class in a name space separated from the name space that the class `SampleLoader` belongs to because the two classes are loaded by different class loaders. Hence, the `MyApp` class cannot directly access the class `SampleLoader`.

3.5 Modifying a system class

The system classes like `java.lang.String` cannot be loaded by a class loader other than the system class loader.

Therefore, `SampleLoader` or `javassist.Loader` shown above cannot modify the system classes at loading time.

If your application needs to do that, the system classes must be *statically* modified. For example, the following program adds a new field `hiddenValue` to `java.lang.String`:

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("java.lang.String");
cc.addField(new CtField(CtClass.intType, "hiddenValue", cc));
cc.writeFile(".");
```

This program produces a file `"./java/lang/String.class"`.

To run your program `MyApp` with this modified `String` class, do as follows:

```
% java -Xbootclasspath/p:. MyApp arg1 arg2...
```

Suppose that the definition of `MyApp` is as follows:

```
public class MyApp {
    public static void main(String[] args) throws Exception {
        System.out.println(String.class.getField("hiddenValue").getName());
    }
}
```

If the modified `String` class is correctly loaded, `MyApp` prints `hiddenValue`.

Note: Applications that use this technique for the purpose of overriding a system class in `rt.jar` should not be deployed as doing so would contravene the Java 2 Runtime Environment binary code license.

3.6 Reloading a class at runtime

If the JVM is launched with the JPDA (Java Platform Debugger Architecture) enabled, a class is dynamically reloadable. After the JVM loads a class, the old version of the class definition can be unloaded and a new one can be reloaded again. That is, the definition of that class can be dynamically modified during runtime. However, the new class definition must be somewhat compatible to the old one. *The JVM does not allow schema changes between the two versions.* They have the same set of methods and fields.

Javassist provides a convenient class for reloading a class at runtime. For more information, see the API documentation of `javassist.tools.HotSwapper`.

[Next page]

---

4. Introspection and customization

`CtClass` provides methods for introspection. The introspective ability of Javassist is compatible with that of the Java reflection API. `CtClass` provides `getName()`, `getSuperclass()`, `getMethods()`, and so on. `CtClass` also provides methods for modifying a class definition. It allows to add a new field, constructor, and method. Instrumenting a method body is also possible.

Methods are represented by `CtMethod` objects. `CtMethod` provides several methods for modifying the definition of the method. Note that if a method is inherited from a super class, then the same `CtMethod` object that represents the inherited method represents the method declared in that super class. A `CtMethod` object corresponds to every method declaration.

For example, if class `Point` declares method `move()` and a subclass `ColorPoint` of `Point` does not override `move()`, the two `move()` methods declared in `Point` and inherited in `ColorPoint` are represented by the identical `CtMethod` object. If the method definition represented by this `CtMethod` object is modified, the modification is reflected on both the methods. If you want to modify only the `move()` method in `ColorPoint`, you first have to add to `ColorPoint` a copy of the `CtMethod` object representing `move()` in `Point`. A copy of the the `CtMethod` object can be obtained by `CtNewMethod.copy()`.

> Javassist does not allow to remove a method or field, but it allows to change the name. So if a method is not necessary any more, it should be renamed and changed to be a private method by calling `setName()` and `setModifiers()` declared in `CtMethod`.
>
> Javassist does not allow to add an extra parameter to an existing method, either. Instead of doing that, a new method receiving the extra parameter as well as the other parameters should be added to the same class. For example, if you want to add an extra `int` parameter `newZ` to a method:
>
> ```
> void move(int newX, int newY) { x = newX; y = newY; }
> ```
>
> in a `Point` class, then you should add the following method to the `Point` class:
>
> ```
> void move(int newX, int newY, int newZ) {
>     // do what you want with newZ.
>     move(newX, newY);
> }
> ```

Javassist also provides low-level API for directly editing a raw class file. For example, `getClassFile()` in `CtClass` returns a `ClassFile` object representing a raw class file. `getMethodInfo()` in `CtMethod` returns a `MethodInfo` object representing a `method_info` structure included in a class file. The low-level API uses the vocabulary from the Java Virtual machine specification. The users must have the knowledge about class files and bytecode. For more details, the users should see the [`javassist.bytecode` package].

The class files modified by Javassist requires the `javassist.runtime` package for runtime support only if some special identifiers starting with `$` are used. Those special identifiers are described below. The class files modified without those special identifiers do not need the `javassist.runtime` package or any other Javassist packages at runtime. For more details, see the API documentation of the `javassist.runtime` package.

4.1 Inserting source text at the beginning/end of a method body

`CtMethod` and `CtConstructor` provide methods `insertBefore()`, `insertAfter()`, and `addCatch()`. They are used for inserting a code fragment into the body of an existing method. The users can specify those code fragments with *source text* written in Java. Javassist includes a simple Java compiler for processing source text. It receives source text written in Java and compiles it into Java bytecode, which will be *inlined* into a method body.

Inserting a code fragment at the position specified by a line number is also possible (if the line number table is contained in the class file). `insertAt()` in `CtMethod` and `CtConstructor` takes source text and a line number in the source file of the original class definition. It compiles the source text and inserts the compiled code at the line number.

The methods `insertBefore()`, `insertAfter()`, `addCatch()`, and `insertAt()` receive a `String` object representing a statement or a block. A statement is a single control structure like `if` and `while` or an expression ending with a semi colon (`;`). A block is a set of statements surrounded with braces `{}`. Hence each of the following lines is an example of valid statement or block:

```
System.out.println("Hello");
{ System.out.println("Hello"); }
if (i < 0) { i = -i; }
```

The statement and the block can refer to fields and methods. They can also refer to the parameters to the method that they are inserted into if that method was compiled with the -g option (to include a local variable attribute in the class file). Otherwise, they must access the method parameters through the special variables `$0`, `$1`, `$2`, ... described below. *Accessing local variables declared in the method is not allowed* although declaring a new local variable in the block is allowed. However, `insertAt()` allows the statement and the block to access local variables if these variables are available at the specified line number and the target method was compiled with the -g option.

The `String` object passed to the methods `insertBefore()`, `insertAfter()`, `addCatch()`, and `insertAt()` are compiled by the compiler included in Javassist. Since the compiler supports language extensions, several identifiers starting with `$` have special meaning:

`$0`, `$1`, `$2`, ...	`this` and actual parameters
`$args`	An array of parameters. The type of `$args` is `Object[]`.
`$$`	All actual parameters. For example, `m($$)` is equivalent to `m($1,$2,`...`)`
`$cflow(`...`)`	`cflow` variable
`$r`	The result type. It is used in a cast expression.
`$w`	The wrapper type. It is used in a cast expression.
`$_`	The resulting value
`$sig`	An array of `java.lang.Class` objects representing the formal parameter types.
`$type`	A `java.lang.Class` object representing the formal result type.

`$class` A `java.lang.Class` object representing the class currently edited.

$0, $1, $2, ...

The parameters passed to the target method are accessible with `$1`, `$2`, ... instead of the original parameter names. `$1` represents the first parameter, `$2` represents the second parameter, and so on. The types of those variables are identical to the parameter types. `$0` is equivalent to `this`. If the method is static, `$0` is not available.

These variables are used as following. Suppose that a class `Point`:

```
class Point {
    int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
}
```

To print the values of `dx` and `dy` whenever the method `move()` is called, execute this program:

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
CtMethod m = cc.getDeclaredMethod("move");
m.insertBefore("{ System.out.println($1); System.out.println($2); }");
cc.writeFile();
```

Note that the source text passed to `insertBefore()` is surrounded with braces `{}`. `insertBefore()` accepts only a single statement or a block surrounded with braces.

The definition of the class `Point` after the modification is like this:

```
class Point {
    int x, y;
    void move(int dx, int dy) {
        { System.out.println(dx); System.out.println(dy); }
        x += dx; y += dy;
    }
}
```

`$1` and `$2` are replaced with `dx` and `dy`, respectively.

`$1`, `$2`, `$3` ... are updatable. If a new value is assigend to one of those variables, then the value of the parameter represented by that variable is also updated.

$args

The variable `$args` represents an array of all the parameters. The type of that variable is an array of class `Object`. If a parameter type is a primitive type such as `int`, then the parameter value is converted into a wrapper object such as `java.lang.Integer` to store in `$args`. Thus, `$args[0]` is equivalent to `$1` unless the type of the first parameter is a primitive type. Note that `$args[0]` is not equivalent to `$0`; `$0` represents `this`.

If an array of `Object` is assigned to `$args`, then each element of that array is assigned to each parameter. If a parameter type is a primitive type, the type of the corresponding element must be a wrapper type. The value is converted from the wrapper type to the primitive type before it is assigned to the parameter.

$$

The variable `$$` is abbreviation of a list of all the parameters separated by commas. For example, if the number of the parameters to method `move()` is three, then

```
move($$)
```

is equivalent to this:

```
move($1, $2, $3)
```

If `move()` does not take any parameters, then `move($$)` is equivalent to `move()`.

`$$` can be used with another method. If you write an expression:

```
exMove($$, context)
```

then this expression is equivalent to:

```
exMove($1, $2, $3, context)
```

Note that `$$` enables generic notation of method call with respect to the number of parameters. It is typically used with `$proceed` shown later.

$cflow

`$cflow` means "control flow". This read-only variable returns the depth of the recursive calls to a specific method.

Suppose that the method shown below is represented by a `CtMethod` object `cm`:

```
int fact(int n) {
    if (n <= 1)
        return n;
    else
        return n * fact(n - 1);
}
```

To use `$cflow`, first declare that `$cflow` is used for monitoring calls to the method `fact()`:

```
CtMethod cm = ...;
cm.useCflow("fact");
```

The parameter to `useCflow()` is the identifier of the declared `$cflow` variable. Any valid Java name can be used as the identifier. Since the identifier can also include `.` (dot), for example, `"my.Test.fact"` is a valid identifier.

Then, `$cflow(fact)` represents the depth of the recursive calls to the method specified by `cm`. The value of `$cflow(fact)` is 0 (zero) when the method is first called whereas it is 1 when the method is recursively called within the method. For example,

```
cm.insertBefore("if ($cflow(fact) == 0)"
              + "    System.out.println(\"fact \" + $1);");
```

translates the method `fact()` so that it shows the parameter. Since the value of `$cflow(fact)` is checked, the method `fact()` does not show the parameter if it is recursively called within `fact()`.

The value of `$cflow` is the number of stack frames associated with the specified method `cm` under the current topmost stack frame for the current thread. `$cflow` is also accessible within a method different from the specified method `cm`.

$r

`$r` represents the result type (return type) of the method. It must be used as the cast type in a cast expression. For example, this is a typical use:

```
Object result = ... ;
$_ = ($r)result;
```

If the result type is a primitive type, then `($r)` follows special semantics. First, if the operand type of the cast expression is a primitive type, `($r)` works as a normal cast operator to the result type. On the other hand, if the operand type is a wrapper type, `($r)` converts from the wrapper type to the result type. For example, if the result type is `int`, then `($r)` converts from `java.lang.Integer` to `int`.

If the result type is `void`, then `($r)` does not convert a type; it does nothing. However, if the operand is a call to a `void` method, then `($r)` results in `null`. For example, if the result type is `void` and `foo()` is a `void` method, then

```
$_ = ($r)foo();
```

is a valid statement.

The cast operator `($r)` is also useful in a `return` statement. Even if the result type is `void`, the following `return` statement is valid:

```
return ($r)result;
```

Here, `result` is some local variable. Since `($r)` is specified, the resulting value is discarded. This `return` statement is regarded as the equivalent of the `return` statement without a resulting value:

```
return;
```

$w

`$w` represents a wrapper type. It must be used as the cast type in a cast expression. `($w)` converts from a primitive type to the corresponding wrapper type. The following code is an example:

```
Integer i = ($w)5;
```

The selected wrapper type depends on the type of the expression following `($w)`. If the type of the expression is `double`, then the wrapper type is `java.lang.Double`.

If the type of the expression following `($w)` is not a primitive type, then `($w)` does nothing.

$_

`insertAfter()` in `CtMethod` and `CtConstructor` inserts the compiled code at the end of the method. In the statement given to `insertAfter()`, not only the variables shown above such as `$0`, `$1`, ... but also `$_` is available.

The variable `$_` represents the resulting value of the method. The type of that variable is the type of the result type (the return type) of the method. If the result type is `void`, then the type of `$_` is `Object` and the value of `$_` is `null`.

Although the compiled code inserted by `insertAfter()` is executed just before the control normally returns from the method, it can be also executed when an exception is thrown from the method. To execute it when an exception is thrown, the second parameter `asFinally` to `insertAfter()` must be `true`.

If an exception is thrown, the compiled code inserted by `insertAfter()` is executed as a `finally` clause. The value of `$_` is `0` or `null` in the compiled code. After the execution of the compiled code terminates, the exception originally thrown is re-thrown to the caller. Note that the value of `$_` is never thrown to the caller; it is rather discarded.

$sig

The value of `$sig` is an array of `java.lang.Class` objects that represent the formal parameter types in declaration order.

$type

The value of `$type` is an `java.lang.Class` object representing the formal type of the result value. This variable refers to `Void.class` if this is a constructor.

$class

The value of `$class` is an `java.lang.Class` object representing the class in which the edited method is declared. This represents the type of `$0`.

addCatch()

`addCatch()` inserts a code fragment into a method body so that the code fragment is executed when the method body throws an exception and the control returns to the caller. In the source text representing the inserted code fragment, the exception value is referred to with the special variable `$e`.

For example, this program:

```
CtMethod m = ...;
CtClass etype = ClassPool.getDefault().get("java.io.IOException");
m.addCatch("{ System.out.println($e); throw $e; }", etype);
```

translates the method body represented by `m` into something like this:

```
try {
    the original method body
}
catch (java.io.IOException e) {
    System.out.println(e);
    throw e;
}
```

Note that the inserted code fragment must end with a `throw` or `return` statement.

4.2 Altering a method body

`CtMethod` and `CtConstructor` provide `setBody()` for substituting a whole method body. They compile the given source text into Java bytecode and substitutes it for the original method body. If the given source text is `null`, the substituted body includes only a `return` statement, which returns zero or null unless the result type is `void`.

In the source text given to `setBody()`, the identifiers starting with `$` have special meaning

`$0`, `$1`, `$2`, ...	`this` and actual parameters
`$args`	An array of parameters. The type of `$args` is `Object[]`.
`$$`	All actual parameters.
`$cflow(`...`)`	`cflow` variable
`$r`	The result type. It is used in a cast expression.
`$w`	The wrapper type. It is used in a cast expression.
`$sig`	An array of `java.lang.Class` objects representing the formal parameter types.
`$type`	A `java.lang.Class` object representing the formal result type.
`$class`	A `java.lang.Class` object representing the class that declares the method currently edited (the type of $0).

Note that `$_` is not available.

Substituting source text for an existing expression

Javassist allows modifying only an expression included in a method body. `javassist.expr.ExprEditor` is a class for replacing an expression in a method body. The users can define a subclass of `ExprEditor` to specify how an expression is modified.

To run an `ExprEditor` object, the users must call `instrument()` in `CtMethod` or `CtClass`. For example,

```
CtMethod cm = ... ;
cm.instrument(
    new ExprEditor() {
        public void edit(MethodCall m)
                      throws CannotCompileException
        {
            if (m.getClassName().equals("Point")
                          && m.getMethodName().equals("move"))
                m.replace("{ $1 = 0; $_ = $proceed($$); }");
        }
    });
```

searches the method body represented by `cm` and replaces all calls to `move()` in class `Point` with a block:

```
{ $1 = 0; $_ = $proceed($$); }
```

so that the first parameter to `move()` is always 0. Note that the substituted code is not an expression but a statement or a block.

The method `instrument()` searches a method body. If it finds an expression such as a method call, field access, and object creation, then it calls `edit()` on the given `ExprEditor` object. The parameter to `edit()` is an object representing the found expression. The `edit()` method can inspect and replace the expression through that object.

Calling `replace()` on the parameter to `edit()` substitutes the given statement or block for the expression. If the given block is an empty block, that is, if `replace("{}")` is executed, then the expression is removed from the method body. If you want to insert a statement (or a block) before/after the expression, a block like the following should be passed to `replace()`:

```
{ before-statements;
  $_ = $proceed($$);
  after-statements; }
```

whichever the expression is either a method call, field access, object creation, or others. The second statement could be:

```
$_ = $proceed();
```

if the expression is read access, or

```
$proceed($$);
```

if the expression is write access.

Local variables available in the target expression is also available in the source text passed to `replace()` if the method searched by `instrument()` was compiled with the -g option (the class file includes a local variable attribute).

javassist.expr.MethodCall

A `MethodCall` object represents a method call. The method `replace()` in `MethodCall` substitutes a statement or a block for the method call. It receives source text representing the substitued statement or block, in which the identifiers starting with `$` have special meaning as in the source text passed to `insertBefore()`.

`$0`	The target object of the method call. This is not equivalent to `this`, which represents the caller-side `this` object. `$0` is `null` if the method is static.
`$1`, `$2`, ...	The parameters of the method call.
`$_`	The resulting value of the method call.
`$r`	The result type of the method call.
`$class`	A `java.lang.Class` object representing the class declaring the method.
`$sig`	An array of `java.lang.Class` objects representing the formal parameter types.
`$type`	A `java.lang.Class` object representing the formal result type.
`$proceed`	The name of the method originally called in the expression.

Here the method call means the one represented by the `MethodCall` object.

The other identifiers such as `$w`, `$args` and `$$` are also available.

Unless the result type of the method call is `void`, a value must be assigned to `$_` in the source text and the type of `$_` is the result type. If the result type is `void`, the type of `$_` is `Object` and the value assigned to `$_` is ignored.

`$proceed` is not a `String` value but special syntax. It must be followed by an argument list surrounded by parentheses `( )`.

javassist.expr.ConstructorCall

A `ConstructorCall` object represents a constructor call such as `this()` and `super` included in a constructor body. The method `replace()` in `ConstructorCall` substitutes a statement or a block for the constructor call. It receives source text representing the substituted statement or block, in which the identifiers starting with `$` have special meaning as in the source text passed to `insertBefore()`.

`$0`	The target object of the constructor call. This is equivalent to `this`.
`$1`, `$2`, ...	The parameters of the constructor call.
`$class`	A `java.lang.Class` object representing the class declaring the constructor.
`$sig`	An array of `java.lang.Class` objects representing the formal parameter types.
`$proceed`	The name of the constructor originally called in the expression.

Here the constructor call means the one represented by the `ConstructorCall` object.

The other identifiers such as `$w`, `$args` and `$$` are also available.

Since any constructor must call either a constructor of the super class or another constructor of the same class, the substituted statement must include a constructor call, normally a call to `$proceed()`.

`$proceed` is not a `String` value but special syntax. It must be followed by an argument list surrounded by parentheses `( )`.

javassist.expr.FieldAccess

A `FieldAccess` object represents field access. The method `edit()` in `ExprEditor` receives this object if field access is found. The method `replace()` in `FieldAccess` receives source text representing the substitued statement or block for the field access.

In the source text, the identifiers starting with `$` have special meaning:

`$0`	The object containing the field accessed by the expression. This is not equivalent to `this`. `this` represents the object that the method including the expression is invoked on. `$0` is `null` if the field is static.
`$1`	The value that would be stored in the field if the expression is write access. Otherwise, `$1` is not available.
`$_`	The resulting value of the field access if the expression is read access. Otherwise, the value stored in `$_` is discarded.
`$r`	The type of the field if the expression is read access. Otherwise, `$r` is `void`.
`$class`	A `java.lang.Class` object representing the class declaring the field.
`$type`	A `java.lang.Class` object representing the field type.
`$proceed`	The name of a virtual method executing the original field access. .

The other identifiers such as `$w`, `$args` and `$$` are also available.

If the expression is read access, a value must be assigned to `$_` in the source text. The type of `$_` is the type of the field.

javassist.expr.NewExpr

A `NewExpr` object represents object creation with the `new` operator (not including array creation). The method `edit()` in `ExprEditor` receives this object if object creation is found. The method `replace()` in `NewExpr` receives source text representing the substitued statement or block for the object creation.

In the source text, the identifiers starting with `$` have special meaning:

`$0`	`null`.
`$1`, `$2`, ...	The parameters to the constructor.
`$_`	The resulting value of the object creation. A newly created object must be stored in this variable.
`$r`	The type of the created object.
`$sig`	An array of `java.lang.Class` objects representing the formal parameter types.
`$type`	A `java.lang.Class` object representing the class of the created object.
`$proceed`	The name of a virtual method executing the original object creation. .

The other identifiers such as `$w`, `$args` and `$$` are also available.

javassist.expr.NewArray

A `NewArray` object represents array creation with the `new` operator. The method `edit()` in `ExprEditor` receives this object if array creation is found. The method `replace()` in `NewArray` receives source text representing the substitued statement or block for the array creation.

In the source text, the identifiers starting with `$` have special meaning:

`$0`	`null`.
`$1`, `$2`, ...	The size of each dimension.

`$_`	The resulting value of the array creation. A newly created array must be stored in this variable.
`$r`	The type of the created array.
`$type`	A `java.lang.Class` object representing the class of the created array.
`$proceed`	The name of a virtual method executing the original array creation. .

The other identifiers such as `$w`, `$args` and `$$` are also available.

For example, if the array creation is the following expression,

```
String[][] s = new String[3][4];
```

then the value of $1 and $2 are 3 and 4, respectively. $3 is not available.

If the array creation is the following expression,

```
String[][] s = new String[3][];
```

then the value of $1 is 3 but $2 is not available.

javassist.expr.Instanceof

A `Instanceof` object represents an `instanceof` expression. The method `edit()` in `ExprEditor` receives this object if an instanceof expression is found. The method `replace()` in `Instanceof` receives source text representing the substitued statement or block for the expression.

In the source text, the identifiers starting with `$` have special meaning:

`$0`	`null`.
`$1`	The value on the left hand side of the original `instanceof` operator.
`$_`	The resulting value of the expression. The type of `$_` is `boolean`.
`$r`	The type on the right hand side of the `instanceof` operator.
`$type`	A `java.lang.Class` object representing the type on the right hand side of the `instanceof` operator.
`$proceed`	The name of a virtual method executing the original `instanceof` expression. It takes one parameter (the type is `java.lang.Object`) and returns true if the parameter value is an instance of the type on the right hand side of the original `instanceof` operator. Otherwise, it returns false.

The other identifiers such as `$w`, `$args` and `$$` are also available.

javassist.expr.Cast

A `Cast` object represents an expression for explicit type casting. The method `edit()` in `ExprEditor` receives this object if explicit type casting is found. The method `replace()` in `Cast` receives source text representing the substitued statement or block for the expression.

In the source text, the identifiers starting with `$` have special meaning:

`$0`	`null`.
`$1`	The value the type of which is explicitly cast.

`$_`	The resulting value of the expression. The type of `$_` is the same as the type after the explicit casting, that is, the type surrounded by `( )`.
`$r`	the type after the explicit casting, or the type surrounded by `( )`.
`$type`	A `java.lang.Class` object representing the same type as `$r`.
`$proceed`	The name of a virtual method executing the original type casting. It takes one parameter of the type `java.lang.Object` and returns it after the explicit type casting specified by the original expression.

The other identifiers such as `$w`, `$args` and `$$` are also available.

javassist.expr.Handler

A `Handler` object represents a `catch` clause of `try-catch` statement. The method `edit()` in `ExprEditor` receives this object if a `catch` is found. The method `insertBefore()` in `Handler` compiles the received source text and inserts it at the beginning of the `catch` clause.

In the source text, the identifiers starting with `$` have meaning:

`$1`	The exception object caught by the `catch` clause.
`$r`	the type of the exception caught by the `catch` clause. It is used in a cast expression.
`$w`	The wrapper type. It is used in a cast expression.
`$type`	A `java.lang.Class` object representing the type of the exception caught by the `catch` clause.

If a new exception object is assigned to `$1`, it is passed to the original `catch` clause as the caught exception.

4.3 Adding a new method or field

Adding a method

Javassist allows the users to create a new method and constructor from scratch. `CtNewMethod` and `CtNewConstructor` provide several factory methods, which are static methods for creating `CtMethod` or `CtConstructor` objects. Especially, `make()` creates a `CtMethod` or `CtConstructor` object from the given source text.

For example, this program:

```
CtClass point = ClassPool.getDefault().get("Point");
CtMethod m = CtNewMethod.make(
                 "public int xmove(int dx) { x += dx; }",
                 point);
point.addMethod(m);
```

adds a public method `xmove()` to class `Point`. In this example, `x` is a `int` field in the class `Point`.

The source text passed to `make()` can include the identifiers starting with `$` except `$_` as in `setBody()`. It can also include `$proceed` if the target object and the target method name are also given to `make()`. For example,

```
CtClass point = ClassPool.getDefault().get("Point");
CtMethod m = CtNewMethod.make(
                 "public int ymove(int dy) { $proceed(0, dy); }",
                 point, "this", "move");
```

this program creates a method `ymove()` defined below:

```
public int ymove(int dy) { this.move(0, dy); }
```

Note that `$proceed` has been replaced with `this.move`.

Javassist provides another way to add a new method. You can first create an abstract method and later give it a method body:

```
CtClass cc = ... ;
CtMethod m = new CtMethod(CtClass.intType, "move",
                          new CtClass[] { CtClass.intType }, cc);
cc.addMethod(m);
m.setBody("{ x += $1; }");
cc.setModifiers(cc.getModifiers() & ~Modifier.ABSTRACT);
```

Since Javassist makes a class abstract if an abstract method is added to the class, you have to explicitly change the class back to a non-abstract one after calling `setBody()`.

Mutual recursive methods

Javassist cannot compile a method if it calls another method that has not been added to a class. (Javassist can compile a method that calls itself recursively.) To add mutual recursive methods to a class, you need a trick shown below. Suppose that you want to add methods `m()` and `n()` to a class represented by `cc`:

```
CtClass cc = ... ;
CtMethod m = CtNewMethod.make("public abstract int m(int i);", cc);
CtMethod n = CtNewMethod.make("public abstract int n(int i);", cc);
cc.addMethod(m);
cc.addMethod(n);
m.setBody("{ return ($1 <= 0) ? 1 : (n($1 - 1) * $1); }");
n.setBody("{ return m($1); }");
cc.setModifiers(cc.getModifiers() & ~Modifier.ABSTRACT);
```

You must first make two abstract methods and add them to the class. Then you can give the method bodies to these methods even if the method bodies include method calls to each other. Finally you must change the class to a not-abstract class since `addMethod()` automatically changes a class into an abstract one if an abstract method is added.

Adding a field

Javassist also allows the users to create a new field.

```
CtClass point = ClassPool.getDefault().get("Point");
CtField f = new CtField(CtClass.intType, "z", point);
point.addField(f);
```

This program adds a field named `z` to class `Point`.

If the initial value of the added field must be specified, the program shown above must be modified into:

```
CtClass point = ClassPool.getDefault().get("Point");
CtField f = new CtField(CtClass.intType, "z", point);
point.addField(f, "0");    // initial value is 0.
```

Now, the method `addField()` receives the second parameter, which is the source text representing an expression computing the initial value. This source text can be any Java expression if the result type of the expression matches the type of the field. Note that an expression does not end with a semi colon (`;`).

Furthermore, the above code can be rewritten into the following simple code:

```
CtClass point = ClassPool.getDefault().get("Point");
CtField f = CtField.make("public int z = 0;", point);
```

```
point.addField(f);
```

Removing a member

To remove a field or a method, call `removeField()` or `removeMethod()` in `CtClass`. A `CtConstructor` can be removed by `removeConstructor()` in `CtClass`.

4.4 Annotations

`CtClass`, `CtMethod`, `CtField` and `CtConstructor` provides a convenient method `getAnnotations()` for reading annotations. It returns an annotation-type object.

For example, suppose the following annotation:

```
public @interface Author {
    String name();
    int year();
}
```

This annotation is used as the following:

```
@Author(name="Chiba", year=2005)
public class Point {
    int x, y;
}
```

Then, the value of the annotation can be obtained by `getAnnotations()`. It returns an array containing annotation-type objects.

```
CtClass cc = ClassPool.getDefault().get("Point");
Object[] all = cc.getAnnotations();
Author a = (Author)all[0];
String name = a.name();
int year = a.year();
System.out.println("name: " + name + ", year: " + year);
```

This code snippet should print:

```
name: Chiba, year: 2005
```

Since the annoation of `Point` is only `@Author`, the length of the array `all` is one and `all[0]` is an `Author` object. The member values of the annotation can be obtained by calling `name()` and `year()` on the `Author` object.

To use `getAnnotations()`, annotation types such as `Author` must be included in the current class path. *They must be also accessible from a `ClassPool` object.* If the class file of an annotation type is not found, Javassist cannot obtain the default values of the members of that annotation type.

4.5 Runtime support classes

In most cases, a class modified by Javassist does not require Javassist to run. However, some kinds of bytecode generated by the Javassist compiler need runtime support classes, which are in the `javassist.runtime` package (for details, please read the API reference of that package). Note that the `javassist.runtime` package is the only package that classes modified by Javassist may need for running. The other Javassist classes are never used at runtime of the modified classes.

4.6 Import

All the class names in source code must be fully qualified (they must include package names). However, the `java.lang` package is an exception; for example, the Javassist compiler can resolve `Object` as well as `java.lang.Object`.

To tell the compiler to search other packages when resolving a class name, call `importPackage()` in `ClassPool`. For example,

```
ClassPool pool = ClassPool.getDefault();
pool.importPackage("java.awt");
CtClass cc = pool.makeClass("Test");
CtField f = CtField.make("public Point p;", cc);
cc.addField(f);
```

The seconde line instructs the compiler to import the `java.awt` package. Thus, the third line will not throw an exception. The compiler can recognize `Point` as `java.awt.Point`.

Note that `importPackage()` *does not* affect the `get()` method in `ClassPool`. Only the compiler considers the imported packages. The parameter to `get()` must be always a fully qualified name.

4.7 Limitations

In the current implementation, the Java compiler included in Javassist has several limitations with respect to the language that the compiler can accept. Those limitations are:

- The new syntax introduced by J2SE 5.0 (including enums and generics) has not been supported. Annotations are supported only by the low level API of Javassist. See the `javassist.bytecode.annotation` package.

- Array initializers, a comma-separated list of expressions enclosed by braces `{` and `}`, are not available unless the array dimension is one.

- Inner classes or anonymous classes are not supported.

- Labeled `continue` and `break` statements are not supported.

- The compiler does not correctly implement the Java method dispatch algorithm. The compiler may confuse if methods defined in a class have the same name but take different parameter lists.

For example,

```
class A {}
class B extends A {}
class C extends B {}

class X {
    void foo(A a) { .. }
    void foo(B b) { .. }
}
```

If the compiled expression is `x.foo(new C())`, where `x` is an instance of X, the compiler may produce a call to `foo(A)` although the compiler can correctly compile `foo((B)new C())`.

- The users are recommended to use `#` as the separator between a class name and a static method or field name. For example, in regular Java,

```
javassist.CtClass.intType.getName()
```

calls a method `getName()` on the object indicated by the static field `intType` in `javassist.CtClass`. In Javassist, the users can write the expression shown above but they are recommended to write:

```
javassist.CtClass#intType.getName()
```

so that the compiler can quickly parse the expression.

Previous page Next page

Previous page

5. Bytecode level API

Javassist also provides lower-level API for directly editing a class file. To use this level of API, you need detailed knowledge of the Java bytecode and the class file format while this level of API allows you any kind of modification of class files.

5.1 Obtaining a `ClassFile` object

A `javassist.bytecode.ClassFile` object represents a class file. To obtian this object, `getClassFile()` in `CtClass` should be called.

Otherwise, you can construct a `javassist.bytecode.ClassFile` directly from a class file. For example,

```
BufferedInputStream fin
    = new BufferedInputStream(new FileInputStream("Point.class"));
ClassFile cf = new ClassFile(new DataInputStream(fin));
```

This code snippet creats a `ClassFile` object from `Point.class`.

A `ClassFile` object can be written back to a class file. `write()` in `ClassFile` writes the contents of the class file to a given `DataOutputStream`.

5.2 Adding and removing a member

`ClassFile` provides `addField()` and `addMethod()` for adding a field or a method (note that a constructor is regarded as a method at the bytecode level). It also provides `addAttribute()` for adding an attribute to the class file.

Note that `FieldInfo`, `MethodInfo`, and `AttributeInfo` objects include a link to a `ConstPool` (constant pool table) object. The `ConstPool` object must be common to the `ClassFile` object and a `FieldInfo` (or `MethodInfo` etc.) object that is added to that `ClassFile` object. In other words, a `FieldInfo` (or `MethodInfo` etc.) object must not be shared among different `ClassFile` objects.

To remove a field or a method from a `ClassFile` object, you must first obtain a `java.util.List` object containing all the fields of the class. `getFields()` and `getMethods()` return the lists. A field or a method can be removed by calling `remove()` on the `List` object. An attribute can be removed in a similar way. Call `getAttributes()` in `FieldInfo` or `MethodInfo` to obtain the list of attributes, and remove one from the list.

5.3 Traversing a method body

To examine every bytecode instruction in a method body, `CodeIterator` is useful. To otbain this object, do as follows:

```
ClassFile cf = ... ;
MethodInfo minfo = cf.getMethod("move");    // we assume move is not overloaded.
CodeAttribute ca = minfo.getCodeAttribute();
CodeIterator i = ca.iterator();
```

A `CodeIterator` object allows you to visit every bytecode instruction one by one from the beginning to the end. The following methods are part of the methods declared in `CodeIterator`:

- `void begin()`
  Move to the first instruction.
- `void move(int index)`
  Move to the instruction specified by the given index.
- `boolean hasNext()`
  Returns true if there is more instructions.
- `int next()`
  Returns the index of the next instruction.
  Note that it does not return the opcode of the next instruction.
- `int byteAt(int index)`
  Returns the unsigned 8bit value at the index.
- `int u16bitAt(int index)`
  Returns the unsigned 16bit value at the index.
- `int write(byte[] code, int index)`
  Writes a byte array at the index.
- `void insert(int index, byte[] code)`
  Inserts a byte array at the index. Branch offsets etc. are automatically adjusted.

The following code snippet displays all the instructions included in a method body:

```
CodeIterator ci = ... ;
while (ci.hasNext()) {
    int index = ci.next();
    int op = ci.byteAt(index);
    System.out.println(Mnemonic.OPCODE[op]);
}
```

5.4 Producing a bytecode sequence

A `Bytecode` object represents a sequence of bytecode instructions. It is a growable array of bytecode. Here is a sample code snippet:

```
ConstPool cp = ...;    // constant pool table
Bytecode b = new Bytecode(cp, 1, 0);
b.addIconst(3);
b.addReturn(CtClass.intType);
CodeAttribute ca = b.toCodeAttribute();
```

This produces the code attribute representing the following sequence:

```
iconst_3
ireturn
```

You can also obtain a byte array containing this sequence by calling `get()` in `Bytecode`. The obtained array can be inserted in another code attribute.

While `Bytecode` provides a number of methods for adding a specific instruction to the sequence, it provides `addOpcode()` for adding an 8bit opcode and `addIndex()` for adding an index. The 8bit value of each opcode is defined in the `Opcode` interface.

`addOpcode()` and other methods for adding a specific instruction are automatically maintain the maximum stack depth unless the control flow does not include a branch. This value can be obtained by calling `getMaxStack()` on the `Bytecode` object. It is also reflected on the `CodeAttribute` object constructed from the `Bytecode` object. To recompute the maximum stack depth of a method body, call `computeMaxStack()` in `CodeAttribute`.

5.5 Annotations (Meta tags)

Annotations are stored in a class file as runtime invisible (or visible) annotations attribute. These attributes can be obtained from `ClassFile`, `MethodInfo`, or `FieldInfo` objects. Call `getAttribute(AnnotationsAttribute.invisibleTag)` on those objects. For more details, see the javadoc manual of `javassist.bytecode.AnnotationsAttribute` class and the `javassist.bytecode.annotation` package.

Javassist also let you access annotations by the higher-level API. If you want to access annotations through `CtClass`, call `getAnnotations()` in `CtClass` or `CtBehavior`.

6. Generics

The lower-level API of Javassist fully supports generics introduced by Java 5. On the other hand, the higher-level API such as `CtClass` does not directly support generics. However, this is not a serious problem for bytecode transformation.

The generics of Java is implemented by the erasure technique. After compilation, all type parameters are dropped off. For example, suppose that your source code declares a parameterized type `Vector<String>`:

```
Vector<String> v = new Vector<String>();
  :
String s = v.get(0);
```

The compiled bytecode is equivalent to the following code:

```
Vector v = new Vector();
  :
String s = (String)v.get(0);
```

So when you write a bytecode transformer, you can just drop off all type parameters. For example, if you have a class:

```
public class Wrapper<T> {
  T value;
  public Wrapper(T t) { value = t; }
}
```

and want to add an interface `Getter<T>` to the class `Wrapper<T>`:

```
public interface Getter<T> {
  T get();
}
```

Then the interface you really have to add is `Getter` (the type parameters `<T>` drops off) and the method you also have to add to the `Wrapper` class is this simple one:

```
public Object get() { return value; }
```

Note that no type parameters are necessary.

7. J2ME

If you modify a class file for the J2ME execution environment, you must perform preverification. Preverifying is basically producing stack maps, which is similar to stack map tables introduced into J2SE at JDK 1.6. Javassist maintains the stack maps for J2ME only if `javassist.bytecode.MethodInfo.doPreverify` is true.

You can also manually produce a stack map for a modified method. For a given method represented by a `CtMethod` object `m`, you can produce a stack map by calling the following methods:

```
m.getMethodInfo().rebuildStackMapForME(cpool);
```

Here, `cpool` is a `ClassPool` object, which is available by calling `getClassPool()` on a `CtClass` object. A `ClassPool` object is responsible for finding class files from given class pathes. To obtain all the `CtMethod` objects, call the `getDeclaredMethods` method on a `CtClass` object.

Previous page