

# *The Binary Component Adaptation User Guide*

---

July 16, 1998

Ralph Keller

---

DISCLAIMER OF WARRANTY. Free of charge Software is provided on an "AS IS" basis, without warranty of any kind, including without limitation the warranties that the Software is free of defects, merchantable, fit for a particular purpose or non-infringing. The entire risk as to the quality and performance of the Software is borne by you.

Copyright (c) 1998 Ralph Keller

ralph@cs.ucsb.edu

All rights reserved.

# 1. Introduction

Binary component adaptation (BCA) [KH98a] is a mechanism that modifies existing components (such as Java class files) to the specific needs of a programmer. Binary component adaptation allows components to be adapted and evolved in binary form and on the fly (during program loading). That is, a programmer can add methods and fields to classes and interfaces, rename and reimplement methods, and alter the type hierarchy even without access to the source code.

This guide focuses on how to install and use the Java implementation of BCA. It does neither explain how BCA works nor how it is implemented. Furthermore, we expect that the reader is familiar with Java and the Java Virtual Machine (JVM).

## 1.1 Related Reading

If you want to know more about the motivation and design rationales of BCA, you should read *Binary Component Adaptation* [KH98a]. Also an interesting paper is *Integrating Independently-Developed Components in Object-Oriented Languages* [Höl93]. If you are interested in the inner workings of BCA, you should have a look at *Implementing Binary Component Adaptation* [KH98b] which describes the implementation for Java.

For further information about BCA, check out the following URL:

<http://www.cs.ucsb.edu/oocsb>

# 2. Restrictions

## 2.1 BCA Restrictions

The current implementation of BCA has some (minor) restrictions:

- *No type checker.* The delta file compiler does not check adaptations for correctness. For example, it is possible to add an interface to a class even though the class does not implement all the required methods. In this case, the adaptation will compile, but at load-time the VM will cause a linkage error.

However, new code added to classes is legal if it compiles correctly and can only reference class members that really exist.

- *No name conflict resolution.* BCA does not (yet) transparently resolve method or field name clashes. For example, if a change introduced by evolution clashes with changes made by adaptation, then a linkage error occurs.
- *Fully qualified type declarations.* The delta file compiler requires all types in methods and field declarations to be fully qualified. For example, it is fine to use the class type `java.lang.String` in a method declaration, but the unqualified `String` would cause a delta file compilation error.

## 2.2 JDK Restrictions

BCA needs to use the class files in `javasrc-1.1.5/classes`. It won't work with the classes provided in the binary distribution of the JDK1.1.5. If the class path is set incorrectly, `bcjava` fails to initialize and aborts silently. Simply set your class path accordingly (e.g., `setenv1 CLASSPATH <bcadir>/javasrc-1.1.5/classes`).

---

<sup>1</sup> The syntax to set environment variables depends on the shell. In the following we assume `tcsh`.

### 3. The Package Structure

The BCA package consists of the following subdirectories:

Directory	Content Description
<code>./bin</code>	Miscellaneous shell scripts and binaries
<code>./lib</code>	Libraries for BCA
<code>./examples</code>	Simple example uses of BCA
<code>./src</code>	Sources for delta file compiler
<code>./javasrc-1.1.5</code>	A stripped version of Sun's JDK1.1.5 source code distribution for SPARC/Solaris that includes the Java VM and javac executables, compiled classes, and libraries. This directory does not contain any source code.

### 4. Installing BCA

1. Untar the release in a directory, so type:

```
tar xvf bca-releasenumber.tar
```

This extracts the package and generates several subdirectories. In the following `<bca-dir>` is the absolute path of the generated bca directory (such as `/users/ralph/bca`)

2. For convenience, you might want to include the directory `<bca-dir>/bin` in your `PATH`.

```
setenv PATH $PATH:<bca-dir>/bin
```

3. Set the class search path to include the classes in `<bca-dir>/src/java` for the delta file compiler such as:

```
setenv CLASSPATH .:<bca-dir>/src/java
```

4. Compile the delta file compiler. Change directory to `<bca-dir>/src/java/dfc/main` and invoke `bcajavac`:

```
javac Main.java
```

### 5. Adaptation Specification

An adaptation specification describes changes to the classes that are retrieved from the file system or network. In general, an adaptation specification includes Java source code fragments for added or reimplemented methods.

The complete grammar for the adaptation specification is described in Appendix A. The following subsections describe the class files specifier and modifications.

**Note:** Currently, each *NameDecl* in an adaptation specification must be *fully qualified*. For example, the type `java.lang.String` is fine but the unqualified type `String` would cause an error.

#### 5.1 Class Files Specifier

The class files specifier defines the set of classes to which the adaptation is applied:

Class Files Specifier	Description
<code>class NameDecl</code>	class file that represents specific class
<code>interface NameDecl</code>	class file that represents specific interface
<code>implements NameDecl</code>	all class files (classes and interfaces) that implement the interface

**Table 1.** Class Files Specifier

## 5.2 Modifications

Currently, the modifications in Table 2 are supported.

	Productions
<i>fields</i>	add field <i>AccessFlagDecls<sub>opt</sub> TypeDecl Identifier</i>
	rename field <i>Identifier</i> to <i>Identifier</i>
	rename reference field <i>NameDecl Identifier</i> to <i>Identifier</i>
<i>methods</i>	add method <i>AccessFlagDecls<sub>opt</sub> TypeDecl Identifier ( FormalParameterListDecl<sub>opt</sub> ) ThrowsDecl<sub>opt</sub> MethodBodyDecl<sub>opt</sub></i>
	rename method <i>TypeDecl Identifier ( FormalParameterListDecl )</i> to <i>Identifier</i>
	rename reference method <i>NameDecl TypeDecl Identifier ( FormalParameterListDecl )</i> to <i>Identifier</i>
<i>interfaces</i>	add implements <i>NameDecl</i>

**Table 2.** Modifications

## 6. Using BCA

The BCA distribution includes three convenient scripts to invoke the interpreter, Java compiler, and delta file compiler.

### 6.1 BCA Interpreter

You invoke the interpreter in the following way:

```
bcajava [-options] class
```

bcajava is a complete replacement for the standard Java interpreter. In addition to the standard options, bcajava recognizes the `-deltas` flag which contains a colon-separated list of delta files that are used for modifications at run-time. Each file name must also include the `.df` extension. For example, a typical invocation of the interpreter looks as following:

```
bcajava -deltas StringEncryption.df:Enumeration.df:ImplementsEnumeration.df Main
```

### 6.2 BCA Java Compiler

To compile against adapted classes, you invoke the Java compiler as following:

```
bcajavac [-options] file.java ...
```

The BCA java compiler is a complete replacement for the JDK's `javac`. In addition to the standard flags, bcajavac recognizes the `-deltas` flag which contains a colon-separated list of delta files used for compilation. Classes that bcajavac loads to look up type information are then modified according to the delta files.

For example, if you have a delta file that adds encryption to `java.lang.String` (such as the `StringEncryption` delta), in order to compile classes that use methods `encrypt` or `decrypt` you have to invoke bcajavac as following:

```
bcajavac -deltas StringEncryption.df UsesAdaptedStringClass.java
```

### 6.3 Delta File Compiler

The delta file compiler translates an adaptation specification into a binary delta file. You invoke delta file compiler as following:

```
bcadfc [-v] file.delta
```

The delta file compiler reads in `file.delta` and, if compilation succeeds, produces a binary delta file. The delta file has the name as specified in the adaptation specification. The `-v` option causes `bcadfc` to print debugging information and the generated delta file.

For example, if you want to compile `StringEncryption.delta`, you invoke the delta file compiler as following:

```
bcadfc StringEncryption.delta
```

This creates the `StringEncryption.df` delta file.

## 6.4 Class File Tools

The BCA distribution also contains three tools that can be helpful when dealing with class files.

### 6.4.1 mainjvm

The tool `mainjvm` is a class file viewer. You invoke it as following:

```
mainjvm classfile.class
```

`mainjvm` shows the complete class file structure in an easy to read form. For example, it displays the constant pool, access modifiers, super class, interfaces, fields, methods, and attributes. Byte codes are shown as hex values.

### 6.4.2 mainjhl

The tool `mainjhl` is a high-level class file viewer. You invoke it as following:

```
mainjhl classfile.class
```

`mainjhl` shows the class in a high-level representation. For example, all references to classes, interfaces, methods, and fields are resolved. `mainjhl` also disassembles byte codes as JVM instructions and resolves all references to the constant pool.

### 6.4.3 mainjdl

`mainjdl` displays a class file after performing modifications. You invoke `mainjdl` as following:

```
mainjdl classfile.class deltafiles
```

`mainjdl` loads the classfile and then makes all modifications described in `deltafiles`, a colon-separated list of delta files such as `StringEncryption.df:Rename.df`.

You can use `mainjdl` to test delta files for their correct behavior. For example, if you run:

```
mainjdl String.class StringEncryption.df
```

Then the printed class file must also include the new methods `encrypt` and `decrypt`:

```
...
61: public encrypt ()Ljava/lang/String;
    0: CodeAttribute: max stack: 4 max locals: 3 codelength: 40
      0x0000: 2a b6 01 12 4c 03 3d a7 00 10 2b 1c 2b 1c 34 10
      0x0010: 56 82 92 55 84 02 01 1c 2a b6 01 13 a1 ff ee bb
      0x0020: 00 11 59 2b b7 00 22 b0
62: public decrypt ()Ljava/lang/String;
    0: CodeAttribute: max stack: 4 max locals: 3 codelength: 40
      0x0000: 2a b6 01 12 4c 03 3d a7 00 10 2b 1c 2b 1c 34 10
      0x0010: 56 82 92 55 84 02 01 1c 2a b6 01 13 a1 ff ee bb
      0x0020: 00 11 59 2b b7 00 22 b0
...
```

## 7. First Experiences with BCA

In this section, we illustrate a simple example of using BCA step by step.

Suppose, we want to extend the standard class `java.lang.String` with functionality for encryption. Using BCA, we write the following adaptation and save it in a file `StringEncryption.delta`:

```
delta StringEncryption adapts class java.lang.String {
  add method public java.lang.String encrypt() {
    char[] buf = this.toCharArray();
    for (int i=0; i<this.length(); i++) {
      buf[i] = (char) (buf[i] ^ 0x56);
    }
    return new String(buf);
  };

  add method public java.lang.String decrypt() {
    char[] buf = this.toCharArray();
    for (int i=0; i<this.length(); i++) {
      buf[i] = (char) (buf[i] ^ 0x56);
    }
    return new String(buf);
  };
}
```

This adaptation adds two methods `encrypt` and `decrypt` to `java.lang.String`. Then we compile the adaptation into the delta file by invoking:

```
bcadfc StringEncryption.delta
```

This generates `StringEncryption.df`. Now, we write a class `Main.java` that uses the encryption functionality provided by the delta.

```
public class Main {
  public static void main(String[] args) {
    String s = "***secret***";

    String encrypted = s.encrypt();
    System.out.println("encrypted: " + encrypted);

    String decrypted = encrypted.decrypt();
    System.out.println("decrypted: " + decrypted);
  }
}
```

We compile `Main.java` by invoking:

```
bcajavac -deltas StringEncryption.df Main.java
```

`bcajavac` requires the `StringEncryption` delta file since we use the methods `encrypt` and `decrypt` in `Main.java`.

Now we can run `Main` by invoking:

```
bcajava -deltas StringEncryption.df Main
```

`Main` produces the following output:

```
encrypted: |||%35$3"|||
decrypted: ***secret***
```

## References

- [GJS96] James Gosling, Bill Joy and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [Höl93] Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. *Proceedings of ECOOP'93*, Springer Verlag LNCS 512, 1993.
- [KH98a] Ralph Keller and Urs Hölzle. Binary Component Adaptation. *Proceedings of ECOOP'98*, Springer Verlag, July 1998.

## Appendix A Adaptation Specification Grammar

The following subsection presents a grammar for the adaptation specification.

### A.1 Grammar Notation

For the grammar, we use the same notation as in chapter 19 of the Java Language Specification [GJS96]. Terminal symbols of the grammar are shown in `fixed-width` font. Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal, followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. Nonterminals that are not specified are identical to the Java Language Specification.

### A.2 Productions from Adaptation Specification

*CompilationUnit*:

`ImportDeclopt delta SimpleName adapts SpecifierDecl UsesDeclopt { ModificationDeclsopt }`

#### A.2.1 Import Declaration

*ImportDecl*:

`SingleTypeImportDecl`

`TypeImportOnDemandDecl`

*SingleTypeImportDecl*:

`import NameDecl`

*TypeImportOnDemandDecl*:

`import NameDecl . *`

#### A.2.2 Class files Specifier Declaration

*SpecifierDecl*:

`ClassSpecifierDecl`

`InterfaceSpecifierDecl`

`ImplementsSpecifierDecl`

*ClassSpecifierDecl*

`class NameDecl`

*InterfaceSpecifierDecl*

`interface NameDecl`

*ImplementsSpecifierDecl*

`implements NameDecl`

#### A.2.3 Uses Declaration

*UsesDecl*:

`uses ClassTypeListDecl`

### A.3 Productions from Modification Declarations

*ModificationDecls*:

`ModificationDecls ModificationDecl ;`

*ModificationDecl:*

*AddFieldDecl*

*AddMethodDecl*

*RenameFieldDecl*

*RenameMethodDecl*

*RenameFieldRefDecl*

*RenameMethodRefDecl*

*AddImplementsDecl*

*AddFieldDecl:*

add field *AccessFlagDecls*<sub>opt</sub> *TypeDecl Identifier*

*AddMethodDecl:*

add method *AccessFlagDecls*<sub>opt</sub> *TypeDecl Identifier* ( *FormalParameterListDecl*<sub>opt</sub> ) *ThrowsDecl*<sub>opt</sub> *MethodBodyDecl*<sub>opt</sub>

*RenameFieldDecl:*

rename field *Identifier* to *Identifier*

*RenameMethodDecl:*

rename method *TypeDecl Identifier* ( *FormalParameterListDecl* ) to *Identifier*

*RenameFieldRefDecl:*

rename reference field *NameDecl Identifier* to *Identifier*

*RenameMethodRefDecl:*

rename reference method *NameDecl TypeDecl Identifier* ( *FormalParameterListDecl* ) to *Identifier*

*AddImplementsDecl:*

add implements *NameDecl*

### **A.3.1 Productions from AddMethodDecl**

*FormalParameterListDecl:*

*ProperFormalParameterListDecl*

*ProperFormalParameterListDecl:*

*ProperFormalParameterListDecl* , *FormalParameterDecl*

*FormalParameterDecl*

*FormalParameterDecl:*

*TypeDecl Identifier*

*ThrowsDecl:*

throws *ClassTypeListDecl*

*ClassTypeListDecl:*

*ClassTypeListDecl* , *ClassTypeDecl*

*ClassTypeDecl*

*ClassTypeDecl:*

*NameDecl*

*MethodBodyDecl:*

{ *BlockDecl* }

*BlockDecl*: any legal Java method body

#### **A.4 Productions from Access Modifiers**

*AccessFlagDecls*:

*AccessFlagDecls* *AccessFlagDecl*

*AccessFlagDecl*: one of

public protected private

static

abstract final native synchronized transient volatile

#### **A.5 Productions from Type Declarations**

*TypeDecl*:

*BaseTypeDecl*

*ReferenceTypeDecl*

*BaseTypeDecl*: one of

byte char double float int long short boolean void

*ReferenceTypeDecl*:

*ArrayTypeDecl*

*ClassOrInterfaceTypeDecl*

*ClassOrInterfaceTypeDecl*:

*NameDecl*

*ArrayTypeDecl*

*BaseTypeDecl* [ ]

*NameDecl* [ ]

*ArrayTypeDecl* [ ]

#### **A.6 Productions from Names**

*NameDecl*:

*SimpleNameDecl*

*QualifiedNameDecl*

*SimpleNameDecl*:

*Identifier*

*QualifiedNameDecl*:

*NameDecl* . *Identifier*