

Bantam Java Runtime System Manual



Marc L. Corliss

Hobart and William Smith Colleges

corliss@hws.edu

<http://math.hws.edu/mcorliss>

E Christopher Lewis

VMware

lewis@vmware.com

<http://www.eclewis.net/>

©2007, Marc L. Corliss and E Christopher Lewis

This manual is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 2.5 License (<http://creativecommons.org/licenses/by-nc-nd/2.5/>). This license allows you to duplicate and distribute this manual in unmodified form for non-commercial purposes. See the license for more details.

Note: this document was extracted from a larger document covering all aspects of the Bantam Java compiler project. It is available at <http://www.bantamjava.com>.

This manual describes the Bantam Java runtime system. The runtime system is an assembly file responsible for handling built-in methods, error handling, and memory management. It is located in the file `/classes/s09/cs433/lib/exceptions.s`. When you run an assembly program in Spim, it is automatically loaded along with the source assembly file. In order for your generated assembly file to work properly (unless you want to write your own runtime system), your compiler must use the same calling conventions and follow a specific format for encoding objects and primitives.

Below we discuss each of the responsibilities of the runtime system, the calling conventions, and the format for encoding objects and primitives.

1 Runtime System Responsibilities

The runtime system contains assembly code for initiating the program, performing memory management, executing the built-in methods (*e.g.*, `Object.clone`, `String.concat`), and for error handling.

Program initiation. The compiled program begins executing in an initial subroutine within the runtime system (`_start`). This subroutine does some initialization of runtime system data structures, and then calls the `main` method within the `Main` class, which is assembly code compiled by your compiler, *i.e.*, it is not part of the runtime system.

Memory management. The runtime system also contains several subroutines for allocating and deallocating memory. Deallocation is done via a simple, mark-and-sweep garbage collector, unless garbage collection is disabled in which case there is no deallocation. None of the methods have to be called by the compiled code, they are called automatically every time an object is created (below, we discuss how to create objects).

Built-in methods. The runtime system also contains the assembly code for every method of a built-in class (methods within `Object`, `Sys`, `String`, and `TextIO`). Your compiler should not generate code for these methods.

Error subroutines. Finally, the runtime system contains subroutines for handling errors. Most error handling subroutines are called automatically by code within the runtime system. There are some that must be called by your compiled code. These are shown in Table 1. For example, if the program attempts to use a `null` object (*e.g.*, dispatch on a `null` object), then the compiled code should call `_null_pointer_error`, which prints an error message and exits.

<code>_null_pointer_error</code>	Null pointer referenced	No additional arguments
<code>_class_cast_error</code>	Class cast error	Object ID passed in \$t0 (Mips), target ID passed in \$t1 (Mips)
<code>_divide_zero_error</code>	Divide (or mod) by zero	No additional arguments

Table 1: Bantam Java runtime system error subroutines.

All of these subroutines take as input the current line number and a pointer to a string representing the filename, so that the line number and filename can be printed in an error message. The line number is passed via the **\$a1** register and the filename pointer is passed via the **\$a2** register. The line number and filename must also be set when dispatching to a method or when constructing an object, since either of these operations could result in a call to an error subroutine.

In addition, the error subroutine **_class_cast_error** takes two additional arguments: a numeric identifier that indicates the type of object being casted and a numeric identifier that indicates the target type in the cast expression. These identifiers are used to print out the types in the error message. The object identifier is passed via register **\$t0** and the target type identifier is passed via register **\$t1**.

2 Calling Conventions

The Bantam Java runtime system uses precise calling conventions that must be followed in order for the compiled program to work correctly. These conventions are discussed below. Note that these conventions only apply to Bantam Java methods. The calling conventions for error subroutines are discussed in the previous section.

Naming. Methods that are compiled to assembly subroutines are named using the format: `<class>.<name>` where `<class>` is the name of the class containing the method and `<name>` is the name of the method. The assembly subroutine for each built-in method follows this convention. So, for example, if you want to call the **clone()** method in the **Object** class at the assembly level, you would call **Object.clone**. The runtime system also assumes that your compiler uses this convention, although the only method it will need to call is **main()** in the **Main** class. It will assume the corresponding compiled subroutine is named **Main.main**. This subroutine label must also be global (using the “.globl Main.main” directive) to allow the runtime system access from another assembly file.

Numeric identifier
Size in bytes
Dispatch table pointer
Value of field 1
Value of field 2
...
Value of field n

Figure 1: Bantam Java encoding of an object with n fields.

Passing the reference object to the callee. Because Bantam Java does not support ‘static’ all method calls are dispatches using a reference object. The reference object is an implicit argument and must be passed to the called method. As in Java, Bantam Java supports only call by value. For objects, it is the reference to the object that is copied not the object itself. This address is passed in the register **\$a0**.

Passing parameters to the callee. All other arguments are passed via the stack. Starting from the leftmost argument and proceeding to the rightmost argument, each actual parameter is evaluated and the result is pushed onto the stack. To push it onto the stack, the value is stored to the address held in the stack pointer and the stack pointer is decremented by one word.

Passing the return value to the caller. The return value (if there is one) is passed via a register. It is passed via register **\$v0**.

3 Encoding Objects and Primitives

To use the runtime system, objects and primitives must be encoded using the format discussed below.

Primitives. There are two types of primitives in Bantam Java: **int** and **boolean**. Nothing special needs to be done for **int** values. To generate a word (in assembly) with the value -10, the compiler would generate the following directive:

```
.word -10
```

Assembly language does not have support for **boolean** values, so **true** and **false** must be encoded using numeric values. **false** is encoded using 0 and **true** is encoded using a non-zero value. Probably the best non-zero value to use for true is -1 (we leave it to the reader to figure out why).

Objects. Figure 1 shows the encoding of an object. An object is encoded as a table with the following entries: (1) a numeric identifier that indicates the class that the object was constructed from, (2) the size of the table in bytes, and (3) a pointer to the dispatch table (which are discussed below). The table also contains an entry per field defined (inherited or otherwise). If there are n fields then there are n words that follow the dispatch table pointer, each of which contains the value for the i th field. If a class has no fields then the dispatch table pointer is the last word in the object. In this case, the size of the object is 12 bytes. In general, for an object with n fields (inherited or otherwise), the size of the object is $12+n*4$.

Order within the list of fields is important. First, the order of fields defined within the same class should be the same as the order that they appeared in the class declaration. More importantly, fields inherited from classes higher in the class hierarchy tree (*i.e.*, closer to **Object**) should appear earlier in the list. If the program accesses a field in an object of type **X**, it should not matter whether the dynamic type of the object is **X** or some subtype of **X**; the field should reside at the same location in the object regardless of the dynamic type. By putting inherited fields earlier in the list, this property is guaranteed. Note: if an inherited field is redeclared, then it is treated as a separate variable, and so it has a separate entry in the list of fields. In this case, the static type of the object will determine which of the two fields is accessed at any point in the program.

Strings. Strings have a slightly different format than other objects. This is because unlike other objects, the size of string objects can vary. The size of the object depends on the length of the sequence of characters. Like other objects, a string object contains an identifier, size, and dispatch table pointer. The fourth entry is the length of the string (in number of characters). Following that is a byte holding each character encoded in ASCII, followed by a null terminating byte (*i.e.*, 0), and possibly followed by other null bytes so that the entire object is word-aligned.

Figure 2(a) shows the assembly code in Mips for encoding the string “abcd”. If the ISA supports the directives “.asciiz” and “.align” (as Mips does), then we can use these to simplify the assembly code as shown in Figure 2(b). The “.asciiz” directive automatically converts each character into the equivalent byte and terminates the string with a null byte. The “.align” directive automatically adds the necessary null bytes to word align the string object.

Class name table. Your compiler will also need to generate a table that the runtime system uses for error reporting. Some error subroutines in the runtime system need to print class names given an object (*e.g.*, on a class cast error). To find the name, it uses a special table called **class.name.table**,

<pre> .word 2 # String identifier .word 24 # size of object in bytes .word String_dispatch_table # pointer to String's # dispatch table .word 4 # length of the string .byte 97 # ASCII encoding for 'a' .byte 98 # ASCII encoding for 'b' .byte 99 # ASCII encoding for 'c' .byte 100 # ASCII encoding for 'd' .byte 0 # null terminating byte .byte 0 # alignment byte .byte 0 # alignment byte .byte 0 # alignment byte </pre>	<pre> .word 2 # String identifier .word 24 # size of object in bytes .word String_dispatch_table # pointer to String's # dispatch table .word 4 # length of the string .asciiz "abcd" # ASCII encoded "abcd", # followed by null byte .align 2 # word align by padding # with null bytes </pre>
---	---

(a)

(b)

Figure 2: A string constant in Mips assembly code: (a) without the “.asciiz” and “.align” directives and (b) with the directives.

which has an entry for each built-in and user-defined class. The table is indexed by the object identifier. Each entry contains a pointer to a string representing the name of the class. Your compiler will have to generate the **class.name.table** as well as the strings for each class name, including strings for built-in classes.

Object references and null. Objects are referred to using their memory address. An object reference (*e.g.*, a variable) is simply a pointer in memory (a word) that contains the memory address of the object that it refers to. If an object reference is **null**, then the pointer contains the value 0. Therefore, testing for **null** is achieved by comparing the reference to 0.

Dispatch tables. For each class, your compiler must generate a dispatch table, including for built-in classes. The dispatch table holds the addresses of every method that is visible from within that particular class. These should be named using the following format:

<class name>_dispatch.table

In particular, the runtime system will need access to **String**'s dispatch table and will expect it to be named **String_dispatch.table**. In addition, the **String_dispatch.table** label will need to be made global (using “.globl”) to allow access from the runtime system.

The dispatch table for each class contains memory addresses to all visible methods (inherited or otherwise). Each word in the dispatch table contains a memory address, specified as a label, to a particular method. As with the list of fields in an object, order within the dispatch table is

important. Methods from classes higher in the class hierarchy tree (*i.e.*, closer to **Object**) should appear earlier in the dispatch table. The order of methods defined within a particular class should be fixed for each dispatch table. In the absence of redefined methods, the order is the same as the order in which the method was defined within the class. If an extended class redefines an inherited method, then the address of the redefined method is placed in the entry that originally contained the inherited method. A new entry is not created. This last requirement is important for handling inheritance and object casting. If an object is casted to a super type, and a method is called that was overridden in the extended class, then this encoding ensures that the overridden method is the method that is called.

Object templates. Objects are not dynamically constructed from scratch. Instead, they are constructed from object templates. For each class, your compiler will generate an object template, including for built-in classes. These should be named using the following format:

<class name>_template

In particular, the runtime system will need access to **String**'s template and **Main**'s template, and will expect these to be named **String_template** and **Main_template**, respectively. In addition, these labels will also need to be made global (using “.globl”) to allow access from the runtime system.

Each template is encoded using the format shown in Figure 1. The templates contain the numeric identifier corresponding to that particular object type, the size of the object in bytes, the pointer to the dispatch table, and the default value for each field in the class (0 for **int**, **false** for **boolean**, **null** otherwise). When a new object is requested (using **new**), the template for the corresponding object type is copied using the built-in subroutine **Object.clone** provided in the runtime library. Therefore, your compiler does not have to explicitly manage memory allocation. Instead, your compiler can use templates and **Object.clone** to construct new objects.

Object initialization. Your compiler will also need to generate subroutines for initializing an object based on the initialization expressions of each field within the object. These should be named using the following format:

<class name>_init

In particular, the runtime system will need access to **Main**'s initialization subroutine in order to create a **Main** object during program initialization. In addition, the **Main_init** label will need to be made global (using “.globl”).

After constructing an object (which is discussed above), the corresponding initialization sub-

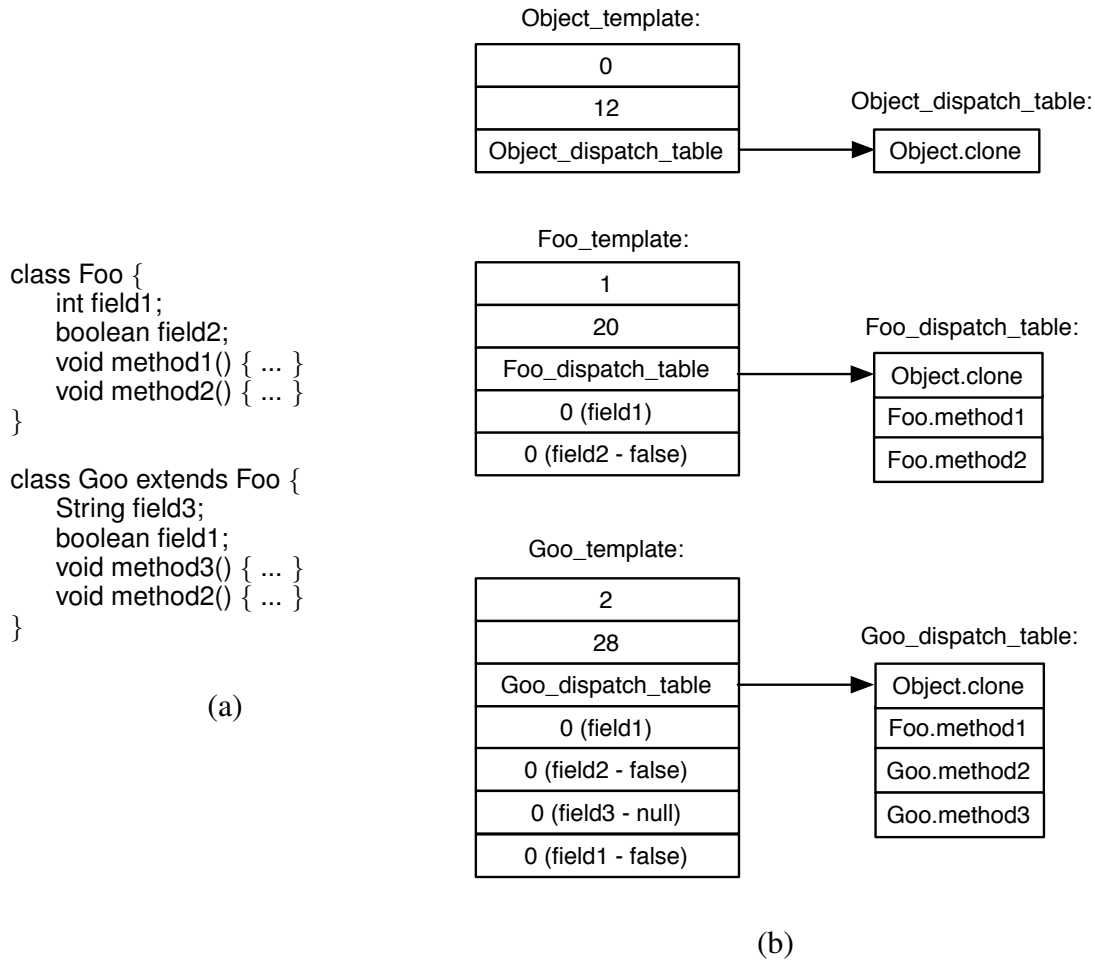


Figure 3: Some example templates and dispatch tables. (a) shows some Bantam Java code (which is incomplete) and (b) shows the templates and dispatch tables for three of the classes: Object, Foo, and Goo.

routine is called. The initialization subroutine will execute each field's initialization expression and assign the resulting value to the corresponding field. Fields should be initialized in the same order in which they are defined within a class. Fields inherited from classes higher in the class hierarchy tree (*e.g.*, closer to **Object**) should be initialized earlier than those from classes lower in the tree. However, an initialization subroutine does not have to directly initialize inherited fields. Instead, it can call the initialization subroutine for the parent class (which can call the initialization subroutine of its parent class, and so forth).

Example. Figure 3(a) shows part of a (contrived) program. Figure 3(b) shows graphically three

of the object templates and three of the dispatch tables that are generated by the compiler (note: other templates and dispatch tables would also be generated). Note that the method **method2** in class **Foo** is overridden in class **Goo**. It appears in the same place in the dispatch table in the **Goo** dispatch table as it does in the **Foo** dispatch table, however, in the **Goo** dispatch table it points to the method in **Goo** rather than in **Foo**. Note also that the field **field1** is redefined in class **Goo**. With fields, the redefined field does not replace the existing field. Instead a new entry is made for the redefined field. When executing code in class **Goo**, the second entry is used when **field1** is referenced. Alternatively, when executing code in class **Foo**, the first entry is used when **field1** is referenced.