2.14 Implementing on Object-Oriented Language

This section is for readers interested in seeing how an **objected-oriented language** like Java executes on a MIPS architecture. It shows the Java bytecodes used for interpretation and the MIPS code for the Java version of some of the C segments in prior sections, including Bubble Sort.

object-oriented language A programming language that is oriented around objects rather than actions, or data versus logic.

Let's quickly review the Java lingo to make sure we are all on the same page. The big idea of object-oriented programming is for programmers to think in terms of abstract objects, and operations are associated with each *type* of object. New types can often be thought of as refinements to existing types, and so some operations for the existing types are used by the new type without change. The hope is that the programmer thinks at a higher level, and that code can be reused more readily if the programmer implements the common operations on many different types.

This different perspective led to a different set of terms. The type of an object is a *class*, which is the definition of a new data type together with the operations that are defined to work on that data type. A particular object is then an *instance* of a class, and creating an object from a class is called *instantiation*. The operations in a class are called *methods*, which are similar to C procedures. Rather than call a procedure as in C, you *invoke* a method in Java. The other members of a class are *fields*, which correspond to variables in C. Variables inside objects are called *instance fields*. Rather than access a structure with a pointer, Java uses an *object reference* to access an object. The syntax for method invocation is `x.y`, where `x` is an object reference and `y` is the method name.

The parent-child relationship between older and newer classes is captured by the verb "extends": a child class *extends* (or subclasses) a parent class. The child class typically will redefine some of the methods found in the parent to match the new data type. There are some methods that work fine, and the child class *inherits* those methods.

To reduce the number of errors associated with pointers and explicit memory deallocation, Java automatically frees unused storage using a separate garbage collector that frees memory when it is full. Hence, `new` creates a new instance of a dynamic object on the heap, but there is no `free` in Java. Java also requires array bounds to be checked at run time to catch another class of errors that can occur in C programs.

Interpreting Java

As mentioned before, Java programs are distributed as Java bytecodes, and the Java Virtual Machine executes Java bytecodes.The JVM understands a binary format called the *class file* format. A class file is a stream of bytes for a single class, containing a table of valid methods with their bytecodes, a pool of constants that acts in part as a symbol table, and other information such as the parent class of this class.

Category	Operation	Java bytecode	Size (bits)	MIPS instr.	Meaning
Arithmetic	add	iadd	8	add	NOS=TOS+NOS; pop
	subtract	isub	8	sub	NOS=TOS-NOS; pop
	increment	iinc I8a I8b	8	addi	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	iload I8/aload I8	16	lw	TOS=Frame[I8]
	load local integer/address	iload_/aload_{0,1,2,3}	8	lw	TOS=Frame[{0,1,2,3}]
	store local integer/address	istore I8/astore I8	16	sw	Frame[I8]=TOS; pop
	load integer/address from array	iaload/ aaload	8	lw	NOS=*NOS[TOS]; pop
	store integer/address into array	iastore/aastore	8	sw	*NNOS[NOS]=TOS; pop2
	load half from array	saload	8	lh	NOS=*NOS[TOS]; pop
	store half into array	sastore	8	sh	*NNOS[NOS]=TOS; pop2
	load byte from array	baload	8	lb	NOS=*NOS[TOS]; pop
	store byte into array	bastore	8	sb	*NNOS[NOS]=TOS; pop2
	load immediate	bipush I8, sipush I16	16, 24	addi	push; TOS=I8 or I16
	load immediate	iconst_{-1,0,1,2,3,4,5}	8	addi	push; TOS={-1,0,1,2,3,4,5}
Logical	and	iand	8	and	NOS=TOS&NOS; pop
	or	ior	8	or	NOS=TOS|NOS; pop
	shift left	ishl	8	sll	NOS=NOS << TOS; pop
	shift right	iushr	8	srl	NOS=NOS >> TOS; pop
Conditional branch	branch on equal	if_icompeq I16	24	beq	if TOS == NOS, go to I16; pop2
	branch on not equal	if_icompne I16	24	bne	if TOS != NOS, go to I16; pop2
	compare	if_icomp{lt,le,gt,ge} I16	24	slt	if TOS {<,<=,>,>=} NOS, go to I16; pop2
Unconditional jump	jump	goto I16	24	j	go to I16
	return	ret, ireturn	8	jr	
	jump to subroutine	jsr I16	24	jal	go to I16; push; TOS=PC+3
Stack management	remove from stack	pop, pop2	8		pop, pop2
	duplicate on stack	dup	8		push; TOS=NOS
	swap top 2 positions on stack	swap	8		T=NOS; NOS=TOS; TOS=T
Safety check	check for null reference	ifnull I16, ifnotnull I16	24		if TOS {==,!=} null, go to I16
	get length of array	arraylength	8		push; TOS = length of array
	check if object a type	instanceof I16	24		TOS = 1 if TOS matches type of Const[I16]; TOS = 0 otherwise
Invocation	invoke method	invokevirtual I16	24		Invoke method in Const[I16], dispatching on type
Allocation	create new class instance	new I16	24		Allocate object type Const[I16] on heap
	create new array	newarray I16	24		Allocate array type Const[I16] on heap

FIGURE 2.14.1 Java bytecode architecture versus MIPS. Although many bytecodes are simple, those in the last half-dozen rows above are complex and specific to Java. Bytecodes are 1 to 5 bytes in length, hence their name. The Java mnemonics use the prefix `i` for 32-bit integer, `a` for reference (address), `s` for 16-bit integers (short), and `b` for 8-bit bytes. We use `I8` for an 8-bit constant and `I16` for a 16-bit constant. MIPS uses registers for operands, but the JVM uses a stack. The compiler knows the maximum size of the operand stack for each method and simply allocates space for it in the current frame. Here is the notation in the Meaning column: TOS: Top Of Stack; NOS: next position below TOS; NNOS: next position below NOS; pop: remove TOS; pop2: remove TOS and NOS; and push: add a position to the stack. *NOS and *NNOS mean access the memory location pointed to by the address in the stack at those positions. Const[] refers to the run time constant pool of a class created by the JVM, and Frame[] refers to the variables of the local method frame. The only missing MIPS instructions from Figure 2.27 are `nor`, `andi`, `ori`, `slti`, and `lui`. The missing bytecodes are a few arithmetic and logical operators, some tricky stack management, compares to 0 and branch, support for branch tables, type conversions, more variations of the complex, Java-specific instructions plus operations on floating-point data, 64-bit integers (longs), and 16-bit characters.

When the JVM is first started, it looks for the class method `main`. To start any Java class, the JVM dynamically loads, links, and initializes a class. The JVM loads a class by first finding the binary representation of the proper class (class file) and creating a class from that binary representation. Linking combines the class into the run-time state of the JVM so that it can be executed. Finally, it executes the class initialization method that is included in every class.

Figure 2.14.1 shows Java bytecodes and their corresponding MIPS instructions, illustrating several differences between the two. First, to simplify compilation, Java uses a stack instead registers for operands. Operands are pushed on the stack, operated on, and then popped off the stack. Second, the designers of the JVM were concerned about code size, so bytecodes vary in length between 1 and 5 bytes versus the 4-byte, fixed-size MIPS instructions. To save space, the JVM even has redundant instructions of different lengths whose only difference is size of the immediate. This decision illustrates a code size variation of our third design principle: make the common case small. Third, the JVM has safety features that are embedded in the architecture. For example, array data transfer instructions check to be sure that the first operand is a reference and that the second index operand is within bounds. Fourth, to allow garbage collectors to find all live pointers, the JVM uses different instructions to operate on addresses versus integers so that the JVM can know what operands contain addresses. MIPS generally collapses integers and addresses together. Finally, unlike MIPS, there are Java-specific instructions that perform complex operations, like allocating an array on the heap or invoking a method.

Compiling a *while* Loop in Java Using bytecodes

EXAMPLE

Compile the *while* loop from page 74, this time using Java bytecodes:

```
while (save[i] == k)
    i += 1;
```

Assume that `i`, `k`, and `save` are the first three local variables. Show the addresses of the bytecodes. The MIPS version of the C loop on page 129 took 6 instructions and 24 bytes. How big is the bytecode version?

ANSWER

The first step is to put the array reference in `save` on the stack:

```
0 aload_3 # Push local variable 3 (save[]) onto stack
```

This 1-byte instruction informs the JVM that an address in local variable 3 is being put on the stack. The 0 on the left of this instruction is the byte address of this first instruction; bytecodes for each method start at 0. The next step is to put the index on the stack:

```
1 iload_1 # Push local variable 1 (i) onto stack
```

Like the prior instruction, this 1-byte instruction is a short version of a more general instruction that takes 2 bytes to load a local variable onto the stack. The next instruction is to get the value from the array element:

```
2 iaload # Put array element (save[i]) onto stack
```

This 1-byte instruction checks the prior two operands, pops them off the stack, and then puts the value of the desired array element onto the new top of the stack. Next, we place `k` on the stack:

```
3 iload_2 # Push local variable 2 (k) onto stack
```

We are now ready for the *while* test:

```
4 if_icompne, Exit # Compare and exit if not equal
```

This 3-byte instruction compares the top two elements of the stack, pops them off the stack, and branches if they are not equal. We are finally ready for the body of the loop:

```
7 iinc, 1, 1 # Increment local variable 1 by 1 (i+=1)
```

This unusual 3-byte instruction increments a local variable by 1 without using the operand stack, an optimization that again saves space. Finally, we return to the top of the loop with a 3-byte jump:

```
10 go to 0 # Go to top of Loop (byte address 0)
```

Thus, the bytecode version takes 7 instructions and 13 bytes, almost half the size of the MIPS C code. (As before, we can optimize this code to jump less; see Exercise 2.14.)

Compiling for Java

Since Java is derived from C and Java has the same built-in types as C, the assignment statement examples in Sections 2.2 to 2.5 are the same in Java as they are in C. The same is true for the *if* statement example in Section 2.6.

The Java version of the *while* loop is different, however. The designers of C leave it up to the programmers to be sure that their code does not exceed the array bounds. The designers of Java wanted to catch array bound bugs, and thus require the compiler to check for such violations. To check bounds, the compiler needs to know what they are. Java includes an extra word in every array that holds the upper bound. The lower bound is defined as 0.

Compiling a *while* Loop in Java

EXAMPLE

Modify the MIPS code for the *while* loop on page 74 to include the array bounds checks that are required by Java. Assume that the length of the array is located just before the first element of the array.

ANSWER

Let's assume that Java arrays reserved the first two words of arrays before the data starts. We'll see the use of the first word soon, but the second word has the array length. Before we enter the loop, let's load the length of the array into a temporary register:

```
lw  $t2,4($s6)  # Temp reg $t2 = length of array save
```

Before we multiply `i` by 4, we must test to see if it's less than 0 or greater than the last element of the array. The first step is to check if `i` is less than 0:

```
Loop:slt   $t0,$s3,$zero # Temp reg $t0 = 1 if i < 0
```

Register `$t0` is set to 1 if `i` is less than 0. Hence, a branch to see if register `$t0` is *not equal to* zero will give us the effect of branching if `i` is less than 0. This pair of instructions, `slt` and `bne`, implements branch on less than. Register `$zero` always contains 0, so this final test is accomplished using the `bne` instruction and comparing register `$t0` to register `$zero`:

```
bne $t0,$zero,IndexOutOfBounds  # if i<0, goto Error
```

Since the array starts at 0, the index of the last array element is one less than the length of the array. Thus, the test of the upper array bound is to be sure that `i` is less than the length of the array. The second step is to set a temporary register to 1 if `i` is less than the array length and then branch to an error if it's not less. That is, we branch to an error if the temporary register is *equal to* zero:

```
slt   $t0,$s3,$t2        # Temp reg $t0 = 0 if i >= length
beq $t0,$zero,IndexOutOfBounds #if i>=length, goto Error
```

Note that these two instructions implement branch on greater than or equal. The next two lines of the MIPS *while* loop are unchanged from the C version:

```
sll  $t1,$s3,2     # Temp reg $t1 = 4 * i
add $t1,$t1,$s6    # $t1 = address of save[i]
```

We need to account for the first 8 bytes that are reserved in Java. We do that by changing the address field of the load from 0 to 8:

```
      lw  $t0,8($t1)     # Temp reg $t0 = save[i]
```

The rest of the MIPS code from the C *while* loop is fine as is:

```
      bne  $t0,$s5, Exit  # go to Exit if save[i] ≠ k
      add  $s3,$s3,1      # i = i + 1
      j    Loop           # go to Loop
Exit:
```

(See Exercise 2.14 for an optimization of this sequence.)

Invoking Methods in Java

The compiler picks the appropriate method depending on the type of the object. In a few cases it is unambiguous, and the method can be invoked with no more overhead than a C procedure. In general, however, the compiler knows only that a given variable contains a pointer to an object that belongs to some subtype of a general class. Since it doesn't know at compile time which subclass the object is, and thus which method should be invoked, the compiler will generate code that first tests to be sure the pointer isn't null, and then uses it to load a pointer to a table with all the legal methods for that type. The first word of the object has the method table address, which is why Java arrays reserve two words. Let's say it's using the fifth method that was declared for that class. (The method order is the same for all subclasses.) The compiler then takes the fifth address from that table and invokes the method at that address.

The cost of object orientation in general is that method invocation includes a conditional branch to be sure that the pointer to the object is valid, a load to get the address of the table of available methods, another load to get the address of the proper method, placing a return address into the return register, and finally a jump register to invoke the method. The next subsection gives a concrete example of method invocation.

A Sort Example in Java

Figure 2.14.2 shows the Java version of exchange sort. A simple difference is that there is no to need to pass the length of the array as a separate parameter since Java arrays include their length: `v.length` denotes the length of `v`.

A more significant difference is that Java methods are prepended with keywords not found in the C procedures. The `sort` method is declared `public`

```
public class sort {
    public static void sort (int[] v) {
      for (int i = 0; i < v.length; i += 1) {
            for (int j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
              swap(v, j);
            }
    }

    protected static void swap(int[] v, int k) {
      int temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
    }}
```

FIGURE 2.14.2 An initial Java procedure that performs a `sort` on the array `v`. Changes from Figure 2.35 are highlighted.

`static` while `swap` is declared `protected static`. **Public** means that `sort` can be invoked from any other method, while **protected** means `swap` can only be called by other methods within the same **package** and from methods within derived classes. A **static method** is another name for a class method—methods that perform class-wide operations and do not apply to an individual object. Static methods are essentially the same as C procedures.

This straightforward translation from C into static methods means there is no ambiguity on method invocation, and so it can be just as efficient as C. It also is limited to sorting integers, which means a different sort has to be written for each data type.

To demonstrate the object orientation of Java, Figure 2.14.3 shows the new version with the changes highlighted. First, we declare `v` to be of the type `Comparable` and replace `v[j] > v[j + 1]` with an invocation of `compareTo`. By changing `v` to this new class, we can use this code to `sort` many data types. The method `compareTo` compares two elements and returns a value greater than 0 if the parameter is larger than the object, 0 if it is equal, and a negative number if it is smaller than the object. These two changes generalize the code so it could sort integers, characters, strings, and so on, provided that there are subclasses of `Comparable` with each of these types and that there is a version of `compareTo` for each type. For pedagogic purposes, we redefine the class `Comparable` and the method `compareTo` here to compare integers. The actual definition of `Comparable` in the Java library is considerably different.

public A Java keyword that allows a method to be invoked by any other method.

protected A Java keyword that restricts invocation of a method to other methods in that package.

package Basically a directory that contains a group of related classes.

static method A method that applies to the whole class rather to an individual object. It is unrelated to static in C.

```
public class sort {
    public static void sort (Comparable[] v) {
     for (int i = 0; i < v.length; i += 1) {
           for (int j = i - 1; j >= 0 && v[j].compareTo(v[j + 1]);
                                          j -= 1) {
             swap(v, j);
           }
    }

    protected static void swap(Comparable[] v, int k) {
     Comparable temp = v[k];
     v[k] = v[k+1];
     v[k+1] = temp;
    }}
public class Comparable {
     public int(compareTo (int x)
     { return value - x; }
     public int value;
}
```

FIGURE 2.14.3 A revised Java procedure that sorts on the array `v` that can take on more types. Changes from Figure 2.14.2 are highlighted.

Starting from the MIPS code that we generated for C, we show what changes we made to create the MIPS code for Java.

For `swap`, the only significant differences are that we must check to be sure the object reference is not null and that each array reference is within bounds. The first test checks that the address in the first parameter is not zero:

```
swap: beq $a0,$zero,NullPointer  #if $a0==0,goto Error
```

Next, we load the length of `v` into a register and check that index `k` is OK.

```
lw $t2,4($a0)     # Temp reg $t2 = length of array v
slt $t0,$a1,$zero # Temp reg $t0 = 1 if k < 0
bne $t0,$zero,IndexOutOfBounds  # if k < 0, goto Error
slt $t0,$a1,$t2   # Temp reg $t0 = 0 if k >= length
beq $t0,$zero,IndexOutOfBounds  #if k>=length,goto Error
```

This check is followed by a check that k+1 is within bounds.

```
addi $t1,$a1,1    # Temp reg $t1 = k+1
slt $t0,$t1,$zero # Temp reg $t0 = 1 if k+1 < 0
bne $t0,$zero,IndexOutOfBounds # if k+1 < 0, goto Error
slt $t0,$t1,$t2   # Temp reg $t0 = 0 if k+1 >= length
beq $t0,$zero,IndexOutOfBounds #if k+1>=length,goto Error
```

Figure 2.14.4 highlights the extra MIPS instructions in swap that a Java compiler might produce. We again must adjust the offset in the load and store to account for two words reserved for the method table and length.

Figure 2.14.5 shows the method body for those new instructions for sort. (We can take the saving, restoring, and return from Figure 2.36.)

The first test is again to make sure the pointer to v is not null:

```
beq $a0,$zero,NullPointer #if $a0==0,goto Error
```

Next we load the length of the array (we use register $s3 to keep it similar to the code for the C version of swap):

```
lw $s3,4($a0)       # $s3 = length of array v
```

Bounds check

```
swap: beq   $a0,$zero,NullPointer       #if $a0==0,goto Error
      lw    $t2,-4($a0)                 # Temp reg $t2 = length of array v
      slt   $t0,$a1,$zero               # Temp reg $t0 = 1 if k < 0
      bne   $t0,$zero,IndexOutOfBounds  # if k < 0, goto Error
      slt   $t0,$a1,$t2                 # Temp reg $t0 = 0 if k >= length
      beq   $t0,$zero,IndexOutOfBounds  # if k >= length, goto Error
      addi  $t1,$a1,1                   # Temp reg $t1 = k+1
      slt   $t0,$t1,$zero               # Temp reg $t0 = 1 if k+1 < 0
      bne   $t0,$zero,IndexOutOfBounds  # if k+1 < 0, goto Error
      slt   $t0,$t1,$t2                 # Temp reg $t0 = 0 if k+1 >= length
      beq   $t0,$zero,IndexOutOfBounds  # if k+1 >= length, goto Error
```

Method body

```
      sll   $t1, $a1, 2           # reg $t1 = k * 4
      add   $t1, $a0, $t1         # reg $t1 = v + (k * 4)
                                  # reg $t1 has the address of v[k]
      lw    $t0, 8($t1)           # reg $t0 (temp) = v[k]
      lw    $t2, 12($t1)          # reg $t2 = v[k + 1]
                                  # refers to next element of v
      sw    $t2, 8($t1)           # v[k] = reg $t2
      sw    $t0, 12($t1)          # v[k+1] = reg $t0 (temp)
```

Procedure return

```
      jr    $ra                   # return to calling routine
```

FIGURE 2.14.4 MIPS assembly code of the procedure swap in Figure 2.33.

Method body				
Move parameters		move	$s2, $a0	# copy parameter $a0 into $s2 (save $a0)
Test ptr null		beq	$a0,$zero,NullPointer	#if $a0==0, goto Error
Get array length		lw	$s3,4($a0)	# $s3 = length of array v
Outer loop		move	$s0, $zero	# i = 0
	for1tst:	slt	$t0, $s0, $s3	# reg $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
		beq	$t0, $zero, exit1	# go to exit1 if $s0 ≥ $s3 (i ≥ n)
Inner loop start		addi	$s1, $s0, -1	# j = i - 1
	for2tst:	slti	$t0, $s1, 0	# reg $t0 = 1 if $s1 < 0 (j < 0)
		bne	$t0, $zero, exit2	# go to exit2 if $s1 < 0 (j < 0)
Test if j too big		slt	$t0,$s1,$s3	# Temp reg $t0 = 0 if j >= length
		beq	$t0,$zero,IndexOutOfBounds	# if j >= length, goto Error
Get v[j]		sll	$t1, $s1, 2	# reg $t1 = j * 4
		add	$t2, $s2, $t1	# reg $t2 = v + (j * 4)
		lw	$t3, 0($t2)	# reg $t3 = v[j]
Test if j+1 < 0 or if j+1 too big		addi	$t1,$s1,1	# Temp reg $t1 = j+1
		slt	$t0,$t1,$zero	# Temp reg $t0 = 1 if j+1 < 0
		bne	$t0,$zero,IndexOutOfBounds	# if j+1 < 0, goto Error
		slt	$t0,$t1,$s3	# Temp reg $t0 = 0 if j+1 >= length
		beq	$t0,$zero,IndexOutOfBounds	# if j+1 >= length, goto Error
Get v[j+1]		lw	$t4, 4($t2)	# reg $t4 = v[j + 1]
Load method table		lw	$t5,0($a0)	# $t5 = address of method table
Get method addr		lw	$t5,8($t5)	# $t5 = address of first method
Pass parameters		move	$a0, $t3	# 1st parameter of compareTo is v[j]
		move	$a1, $t4	# 2nd param. of compareTo is v[j+1]
Set return addr		la	$ra,L1	# load return address
Call indirectly		jr	$t5	# call code for compareTo
Test if should skip swap	L1:	slt	$t0, $zero, $v0	# reg $t0 = 0 if 0 ≥ $v0
		beq	$t0, $zero, exit2	# go to exit2 if $t4 ≥ $t3
Pass parameters and call swap		move	$a0, $s2	# 1st parameter of swap is v
		move	$a1, $s1	# 2nd parameter of swap is j
		jal	swap	# swap code shown in Figure 2.34
Inner loop end		addi	$s1, $s1, -1	# j -= 1
		j	for2tst	# jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1	# i += 1
		j	for1tst	# jump to test of outer loop

FIGURE 2.14.5 MIPS assembly version of the method body of the Java version of `sort`. The new code is highlighted in this figure. We must still add the code to save and restore registers and the return from the MIPS code found in Figure 2.36. To keep the code similar to that figure, we load `v.length` into `$s3` instead of into a temporary register. To reduce the number of lines of code, we make the simplifying assumption that `compareTo` is a leaf procedure and we do not need to push registers to be saved on the stack.

Now we must ensure that the index is within bounds. Since the first test of the inner loop is to test if `j` is negative, we can skip that initial bound test. That leaves the test for too big:

```
slt $t0,$s1,$s3        # Temp reg $t0 = 0 if j >= length
beq $t0,$zero,IndexOutOfBounds #if j>=length, goto Error
```

The code for testing `j + 1` is quite similar to the code for checking `k + 1` in `swap`, so we skip it here.

The key difference is the invocation of `compareTo`. We first load the address of the table of legal methods, which we assume is two words before the beginning of the array:

```
lw $t5,0($a0)       # $t5 = address of method table
```

Given the address of the method table for this object, we then get the desired method. Let's assume `compareTo` is the third method in the `Comparable` class. To pick the address of the third method, we load that address into a temporary register:

```
lw $t5,8($t5)       # $t5 = address of third method
```

We are now ready to call `compareTo`. The next step is to save the necessary registers on the stack. Fortunately, we don't need the temporary registers or argument registers after the method invocation, so there is nothing to save. Thus, we simply pass the parameters for `compareTo`:

```
move $a0, $t3    # 1st parameter of compareTo is v[j]
move $a1, $t4    # 2nd parameter of compareTo is v[j+1]
```

Since we are using a jump register to invoke `compareTo`, we need to explicitly pass the return address. We use the pseudoinstruction load address (`la`) and label where we want to return, and then do the indirect jump:

```
la $ra,L1            # load return address
jr $t5               # to code for compareTo
```

The method returns with `$v0` determining which is larger. If `$v0` > 0, then `v[j]` >`v[j+1]`, and we need to `swap`. Thus to skip the `swap`, we need to test if `$v0` ≤ 0, which is the same as 0 ≥ `$v0`. We also need to include the label for the return address:

```
L1: slt $t0, $zero, $v0   # reg $t0 = 0 if 0 ≥ $v0
    beq $t0, $zero, exit2 # go to exit2 if v[j+1] ≥ v[j]
```

The MIPS code for `compareTo` is left as an exercise (see Exercise 2.48).

Hardware Software Interface

The main changes for the Java version of `sort` and `swap` are testing for null object references and index out-of-bounds errors, and the extra method invocation to give a more general compare. This method invocation is more expensive than a C procedure call, since it requires a load, conditional branch, a pair of chained loads, and an indirect jump. As we will see in later chapters, chained loads and indirect jumps can be relatively slow on modern processors. The increasing popularity of Java suggests that many programmers today are willing to leverage the high performance of modern processors to pay for error checking and code reuse.

Elaboration: Although we test each reference to `j` and `j + 1` to be sure that these indices are within bounds, an assembly language programmer might look at the code and reason as follows:

1. The inner for loop is only executed if `j ≥ 0` and since `j + 1` > *j*, there is no need to test `j+1` to see if it is less than 0.
2. Since `i` takes on the values, 0, 1, 2, ..., (`data.length` – 1) and since j takes on the values `i`–1, `i`–2, ..., 2, 1, 0, there is no need to test if `j ≥ datalength` since the largest value `j` can be is `data.length` – 2.
3. Following the same reasoning, there is no need to test if `j + 1 ≥ datalength` since the largest value of `j`+1 is `data.length` – 1.

We will see some coding tricks in Chapter 3 and superscalar execution in Chapter 6 that lower the effective cost of such bounds checking, but only high optimizing compilers can reason this way. Note that if the compiler inlined the `swap` method into `sort`, many checks would be unnecessary.

Elaboration: Look carefully at the code for `swap` in Figure 2.14.4. See anything wrong in the code, or at least, in the explanation of how the code works? It implicitly assumes that each `Comparable` element in `v` is 4 bytes long. Surely you need much more than 4 bytes for a complex subclass of `Comparable`, which could contain any number of fields? Surprisingly, this code does work, because an important property of Java's semantics basically forces the use of the same, small representation for all variables, fields, and array elements that belong to `Comparable` or its subclasses.

Java types are divided into *primitive types*—the predefined types for numbers, characters, and Booleans—and *reference types*—the built-in classes like `String`, user-defined classes, and arrays. Values of reference types are pointers (also called *references*) to anonymous objects that are themselves allocated in the heap. For the programmer, this means that assigning one variable to another does not create a new object, but instead makes both variables refer to the same object. Because these objects are anonymous and programs therefore have no way to refer to them directly, a program must use indirection through a variable to read or write any objects' fields (vari-

ables). Thus, the data structure allocated for the array `v` consists entirely of pointers, it is safe to assume they are all the same size, and the same swapping code works for all of `Comparable`'s subtypes.

To write sorting and swapping functions for arrays of primitive types requires that we write new versions of the functions, one for each type. This replication is for two reasons. First, primitive type values do not include the references to dispatching tables that we used on `Comparables` to determine at run time how to compare values. Second, primitive values come in different sizes: 1, 2, 4, or 8 bytes.

The pervasive use of pointers in Java is elegant in its consistency, with the penalty being a level of indirection and a requirement that objects be allocated on the heap. Furthermore, in any language where the lifetimes of the heap-allocated anonymous objects are independent of the lifetimes of the named variables, fields, and array elements that reference them, programmers must deal with the problem of deciding when it is safe to deallocate heap-allocated storage. Java's designers chose to use garbage collection. Of course, use of garbage collection rather than explicit user memory management also improves program safety.

C++ provides an interesting contrast. Although one can write essentially the same pointer-manipulating solution in C++, there is another option. In C++, one can elect to forgo the level of indirection and manipulate an array of objects directly, rather than an array of pointers to those objects. To do so, C++ programmers would typically use the template capability, which allows a class or function to be parameterized by the *type* of data it acts on. Templates, however, are compiled using the equivalent of macro expansion. That is, if we declared an instance of `sort` capable of sorting types `X` and `Y`, C++ would create two copies of the code for the class: one for `sort<X>` and one for `sort<Y>`, each specialized accordingly. This solution increases code size in exchange for making comparison faster (since the function calls would not be indirect, and might even be subject to inline expansion). Of course, the speed advantage would be canceled if swapping the objects required moving large amounts of data instead of just single pointers. As always, the best design depends on the details of the problem.