TABLE 3.1: *Mnemonic type letters*

Letter	Type
a	reference
b	byte or boolean
c	char
d	double
f	float
i	int
l	long
s	short

letter of the mnemonic often tells you what type the mnemonic operates on. For example, `iconst_1` loads an `int` 1 onto the stack, while `lconst_1` loads a `long` 1 onto the stack. The letter `i` indicates an `int`, and the letter `l` indicates a `long`. Table 3.1 summarizes the mnemonic naming conventions.

Do not be misled by the naming convention into believing in mnemonics that do not exist. Each mnemonic corresponds to a number between 0 and 255 in the `class` file. This number is called an *opcode*. Just because there's an opcode for the mnemonic `iand` to compute the bitwise `and` of two `int`s, it doesn't mean that there's an opcode for the mnemonic `dand` to compute the bitwise `and` of two `double`s. The complete list of mnemonics can be found in appendix A.

3.4 Testing Code Examples

Code examples appear throughout this chapter. Unfortunately, we haven't really discussed how to do output, so it will be a little tricky to actually try out any of these code examples. Output involves some mucking about with method calls and sometimes with creating new objects and other such complicated things that are better left for a later chapter.

The good news is that you can use Java to assemble a test harness for these code samples. You can wrap the test code up in a small, easy-to-write class, then use Java to perform the input and output. For example, here is some code to answer the question, "What do you get if you multiply 6 by 9?"

```
bipush 6        ; Push 6
bipush 9        ; Push 9
imul            ; Result is 54
```