Figure 13.1 Method overloading.

The example started on this slide will be refined over the next few slides, and will help us to introduce you to one of the most fundamental concepts in object-oriented programming.

Suppose we wish to write a program that computes statistics from both numerical (i.e. 65, 94, etc.) and letter (i.e. C, A+, etc.) grades. First, let's see if a grade is a pass:

```
System.out.print("letter grade? (y/n) ");
if (keyb.readLine().equals("y")) {
    String mark = keyb.readLine():
    if (isPass(mark))
        System.out.println("pass");
}
else {    // numerical grade
    int mark = Integer.parseInt(keyb.readLine());
    if (isPass(mark))
        System.out.println("pass");
}
```

In order for this to work, we need two **isPass()** methods—one taking a **String** argument and the other taking an **int** argument.

```
private static boolean isPass (int mark) {
    return mark >= PASS_NUMBER;    // PASS_NUMBER is 50
}
private static boolean isPass (String mark) {
    return mark.compareTo(PASS_LETTER) >= 0;    // PASS_LETTER is D-
}
```

The overloaded **isPass()** method is handy, but we still need to keep track of which marks are numerical and which are not.

Consider this: If we stored our list of marks as a Vector (which is a good choice—why?), then when it comes time to recover the marks (for example, we might want to calculate the average), how do we know which are numerical and which are not?

Here is one way, but it's not very elegant:

Exercise: Can you think of a better way?