Applying the inductive hypothesis, we obtain
\[
\sum_{i=1}^{N+1} i^2 = \frac{N(N+1)(2N+1)}{6} + (N+1)^2
\]
\[
= (N+1) \left[ \frac{N(2N+1)}{6} + (N+1) \right]
\]
\[
= (N+1) \frac{2N^2 + 7N + 6}{6}
\]
\[
= \frac{(N+1)(N+2)(2N+3)}{6}
\]

Thus,
\[
\sum_{i=1}^{N+1} i^2 = \frac{(N+1)[(N+1)+1][2(N+1)+1]}{6}
\]

proving the theorem.

**Proof by Counterexample**
The statement \( F_k \leq k^2 \) is false. The easiest way to prove this is to compute \( F_{11} = 144 > 11^2 \).

**Proof by Contradiction**
Proof by contradiction proceeds by assuming that the theorem is false and showing that this assumption implies that some known property is false, and hence the original assumption was erroneous. A classic example is the proof that there is an infinite number of primes. To prove this, we assume that the theorem is false, so that there is some largest prime \( P_k \). Let \( P_1, P_2, \ldots, P_k \) be all the primes in order and consider
\[
N = P_1P_2P_3 \cdots P_k + 1
\]

Clearly, \( N \) is larger than \( P_k \), so by assumption \( N \) is not prime. However, none of \( P_1, P_2, \ldots, P_k \) divides \( N \) exactly, because there will always be a remainder of 1. This is a contradiction, because every number is either prime or a product of primes. Hence, the original assumption, that \( P_k \) is the largest prime, is false, which implies that the theorem is true.

### 1.3 A Brief Introduction to Recursion

Most mathematical functions that we are familiar with are described by a simple formula. For instance, we can convert temperatures from Fahrenheit to Celsius by applying the formula
\[
C = \frac{5(F - 32)}{9}
\]

Given this formula, it is trivial to write a Java method; with declarations and braces removed, the one-line formula translates to one line of Java.

Mathematical functions are sometimes defined in a less standard form. As an example, we can define a function \( f \), valid on nonnegative integers, that satisfies \( f(0) = 0 \) and
public static int f(int x) {
    if (x == 0) return 0;
    else return 2 * f(x - 1) + x * x;
}

Figure 1.2 A recursive method

\[ f(x) = 2f(x - 1) + x^2. \] From this definition we see that \( f(1) = 1, f(2) = 6, f(3) = 21, \) and \( f(4) = 88 \). A function that is defined in terms of itself is called recursive. Java allows functions to be recursive.\(^1\) It is important to remember that what Java provides is merely an attempt to follow the recursive spirit. Not all mathematically recursive functions are efficiently (or correctly) implemented by Java's simulation of recursion. The idea is that the recursive function \( f \) ought to be expressible in only a few lines, just like a nonrecursive function. Figure 1.2 shows the recursive implementation of \( f \).

Lines 3 and 4 handle what is known as the base case, that is, the value for which the function is directly known without resorting to recursion. Just as declaring \( f(x) = 2f(x - 1) + x^2 \) is meaningless, mathematically, without including the fact that \( f(0) = 0 \), the recursive Java method doesn't make sense without a base case. Line 6 makes the recursive call.

There are several important and possibly confusing points about recursion. A common question is: Isn't this just circular logic? The answer is that although we are defining a method in terms of itself, we are not defining a particular instance of the method in terms of itself. In other words, evaluating \( f(5) \) by computing \( f(5) \) would be circular. Evaluating \( f(5) \) by computing \( f(4) \) is not circular—unless, of course, \( f(4) \) is evaluated by eventually computing \( f(5) \). The two most important issues are probably the how and why questions. In Chapter 3, the how and why issues are formally resolved. We will give an incomplete description here.

It turns out that recursive calls are handled no differently from any others. If \( f \) is called with the value of 4, then line 6 requires the computation of \( 2 * f(3) + 4 \). Thus, a call is made to compute \( f(3) \). This requires the computation of \( 2 * f(2) + 3 \). Therefore, another call is made to compute \( f(2) \). This means that \( 2 * f(1) + 2 * 2 \) must be evaluated. To do so, \( f(1) \) is computed as \( 2 * f(0) + 1 \). Now, \( f(0) \) must be evaluated. Since this is a base case, we know a priori that \( f(0) = 0 \). This enables the completion of the calculation for \( f(1) \), which is now seen to be 1. Then \( f(2), f(3), \) and finally \( f(4) \) can be determined. All the bookkeeping needed to keep track of pending calls (those started but waiting for a recursive call to complete), along with their variables, is done by the computer automatically.

An important point, however, is that recursive calls will keep on being made until a base case is reached. For instance, an attempt to evaluate \( f(-1) \) will result in calls to \( f(-2), f(-3), \) and so on; the answer is not made. (It even is by line 6, to be 1.) The computer will eventually, its behavior is abnormal. Given an input evaluated either as positive or as nonnegative, the computer will always return a correct value. This is called a special case.

These consist of

1. Base cases.
3. Always be able to test.

Throughout this book, we assume that other words, so we might have some of those, is finite, eventually in some definiteness. Our only definition that some word is recursive.

Our recursive word, then we have all the words in means by recur: if the dictionary circularly defines...
public static int bad( int n )
{
    if( n == 0 )
        return 0;
    else
        return bad( n / 3 + 1 ) + n - 1;
}

Figure 1.3 A nonterminating recursive method

$f(-3)$, and so on. Since this will never get to a base case, the program won't be able to compute the answer (which is undefined anyway). Occasionally, a much more subtle error is made, which is exhibited in Figure 1.3. The error in Figure 1.3 is that bad(1) is defined, by line 6, to be bad(1). Obviously, this doesn't give any clue as to what bad(1) actually is. The computer will thus repeatedly make calls to bad(1) in an attempt to resolve its values. Eventually, its bookkeeping system will run out of space, and the program will terminate abnormally. Generally, we would say that this method doesn't work for one special case but is correct otherwise. This isn't true here, since bad(2) calls bad(1). Thus, bad(2) cannot be evaluated either. Furthermore, bad(3), bad(4), and bad(5) all make calls to bad(2). Since bad(2) is unevaluable, none of these values are either. In fact, this program doesn't work for any nonnegative value of n, except 0. With recursive programs, there is no such thing as a "special case."

These considerations lead to the first two fundamental rules of recursion:

1. Base cases. You must always have some base cases, which can be solved without recursion.
2. Making progress. For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case.

Throughout this book, we will use recursion to solve problems. As an example of a nonmathematical use, consider a large dictionary. Words in dictionaries are defined in terms of other words. When we look up a word, we might not always understand the definition, so we might have to look up words in the definition. Likewise, we might not understand some of those, so we might have to continue this search for a while. Because the dictionary is finite, eventually either (1) we will come to a point where we understand all of the words in some definition (and thus understand that definition and retrace our path through the other definitions) or (2) we will find that the definitions are circular and we are stuck, or that some word we need to understand for a definition is not in the dictionary.

Our recursive strategy to understand words is as follows: If we know the meaning of a word, then we are done; otherwise, we look the word up in the dictionary. If we understand all the words in the definition, we are done; otherwise, we figure out what the definition means by recursively looking up the words we don't know. This procedure will terminate if the dictionary is well defined but can loop indefinitely if a word is either not defined or circularly defined.
Figure 1.4 Recursive routine to print an integer

Printing Out Numbers
Suppose we have a positive integer, \( n \), that we wish to print out. Our routine will have the
heading printOut(\( n \)). Assume that the only I/O routines available will take a single-digit
number and output it to the terminal. We will call this routine printDigit; for example,
printDigit(4) will output a 4 to the terminal.

Recursion provides a very clean solution to this problem. To print out 76234, we need
to first print out 7623 and then print out 4. The second step is easily accomplished with
the statement printDigit(n\%10), but the first doesn’t seem any simpler than the original
problem. Indeed it is virtually the same problem, so we can solve it recursively with the
statement printOut(n/10).

This tells us how to solve the general problem, but we still need to make sure that
the program doesn’t loop indefinitely. Since we haven’t defined a base case yet, it is clear
that we still have something to do. Our base case will be printDigit(n) if 0 \( \leq n < 10 \). Now
printOut(n) is defined for every positive number from 0 to 9, and larger numbers are defined
in terms of a smaller positive number. Thus, there is no cycle. The entire method is shown
in Figure 1.4.

We have made no effort to do this efficiently. We could have avoided using the mod
routine (which can be very expensive) because \( n \% 10 = n - \lfloor n/10 \rfloor \times 10 \).

Recursion and Induction
Let us prove (somewhat) rigorously that the recursive number-printing program works. To
do so, we’ll use a proof by induction.

Theorem 1.4.
The recursive number-printing algorithm is correct for \( n \geq 0 \).

Proof (by induction on the number of digits in \( n \)).
First, if \( n \) has one digit, then the program is trivially correct, since it merely makes
a call to printDigit. Assume then that printOut works for all numbers of \( k \) or fewer
digits. A number of \( k + 1 \) digits is expressed by its first \( k \) digits followed by its least
significant digit. But the number formed by the first \( k \) digits is exactly \( \lfloor n/10 \rfloor \), which,
by the inductive hypothesis, is correctly printed, and the last digit is \( n \mod 10 \), so
the program prints out any \( (k + 1) \)-digit number correctly. Thus, by induction, all numbers
are correctly printed.

\[ [x] \] is the largest integer that is less than or equal to \( x \).

1.4 Imp

An important go
tant mechanism is identica
e to describe the logi
could be used.
This proof probably seems a little strange in that it is virtually identical to the algorithm description. It illustrates that in designing a recursive program, all smaller instances of the same problem (which are on the path to a base case) may be assumed to work correctly. The recursive program needs only to combine solutions to smaller problems, which are “magically” obtained by recursion, into a solution for the current problem. The mathematical justification for this is proof by induction. This gives the third rule of recursion:

3. **Design rule.** Assume that all the recursive calls work.

This rule is important because it means that when designing recursive programs, you generally don’t need to know the details of the bookkeeping arrangements, and you don’t have to try to trace through the myriad of recursive calls. Frequently, it is extremely difficult to track down the actual sequence of recursive calls. Of course, in many cases this is an indication of a good use of recursion, since the computer is being allowed to work out the complicated details.

The main problem with recursion is the hidden bookkeeping costs. Although these costs are almost always justifiable, because recursive programs not only simplify the algorithm design but also tend to give cleaner code, recursion should never be used as a substitute for a simple for loop. We’ll discuss the overhead involved in recursion in more detail in Section 3.6.

When writing recursive routines, it is crucial to keep in mind the four basic rules of recursion:

1. **Base cases.** You must always have some base cases, which can be solved without recursion.
2. **Making progress.** For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case.
3. **Design rule.** Assume that all the recursive calls work.
4. **Compound interest rule.** Never duplicate work by solving the same instance of a problem in separate recursive calls.

The fourth rule, which will be justified (along with its nickname) in later sections, is the reason that it is generally a bad idea to use recursion to evaluate simple mathematical functions, such as the Fibonacci numbers. As long as you keep these rules in mind, recursive programming should be straightforward.

### 1.4 Implementing Generic Components Pre Java 5

An important goal of object-oriented programming is the support of code reuse. An important mechanism that supports this goal is the **generic mechanism**: If the implementation is identical except for the basic type of the object, a **generic implementation** can be used to describe the basic functionality. For instance, a method can be written to sort an array of items; the logic is independent of the types of objects being sorted, so a generic method could be used.