11.3 Recursive Queries

Suppose that we are given the following recursive definition of a sequence

$$\begin{aligned} t_1 &= 2 \\ t_n &= 3t_{n-1} - 1, \ n > 1 \end{aligned}$$

We can express such a sequence using function notation if we write

$$\begin{aligned} t(1) &= 2 \\ t(n) &= 3t(n-1) - 1, \ n > 1 \end{aligned}$$

To implement such a function with a Java method is simple. (The tricky part is understanding how the method operates.)

Example 1

The following method returns the value of the function defined by

$$\begin{aligned} t(1) &= 2 \\ t(n) &= 3t(n-1) - 1, \ n > 1 \end{aligned}$$

If the method is given an invalid value of the parameter, `n`, it throws an exception.

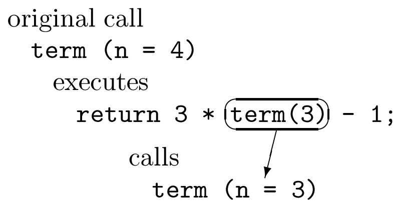
```
public static int term (int n)
{
  if (n < 1)
    throw new RuntimeException("Invalid parameter");
  else if (n == 1)
    return 2;
  else
    return 3 * term(n-1) - 1;
}
```

At first glance, the method may not appear to do anything; it seems to be nothing more than a definition of the sequence. To see how it works, suppose that `term` is called with `n` $= 4$. Since `n` > 1, the method executes the statement

```
return 3 * term(3) - 1;
```

The right side of this assignment statement involves a call to the method `term`. We have seen methods call other methods before. In such a case, execution of the first method is suspended until the called method returns the value it has been asked to compute. The same thing happens here; the only difference is that here both the calling method and the called method are the same. To visualize the process, we can think of a new copy of `term` being created by the call. This copy is identical to the original but it has its own copy of the parameter `n`. This version of the parameter `n` has the value 3. If the method had any local variables, there would also be separate copies of them in both the original and the called versions of the method.

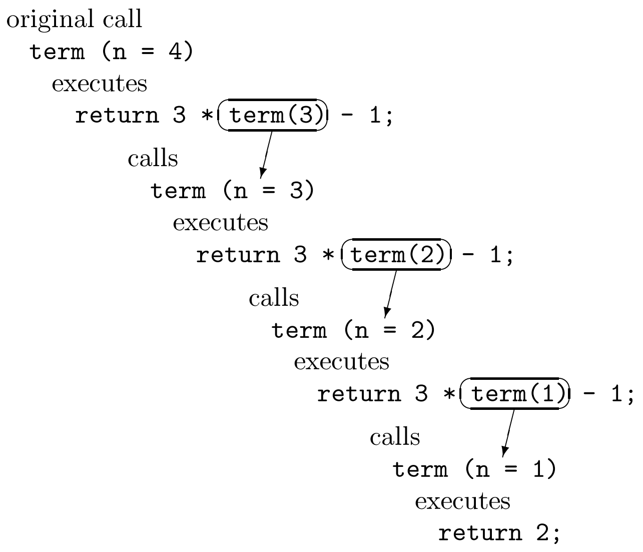
We can illustrate this process using a diagram.



This copy of `term` (with `n = 3`) now executes the statement

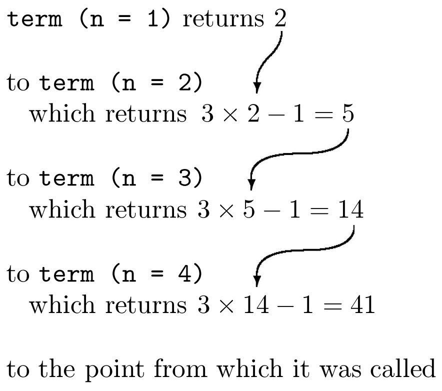
```
term = 3 * term(2) - 1;
```

Before it can complete execution of this statement, it must call another copy of `term`, this time with `n = 2`. The process continues until a copy is called with `n = 1`, as shown in the next diagram.



```
original call
  term (n = 4)
    executes
      return 3 * term(3) - 1;
          calls
            term (n = 3)
              executes
                return 3 * term(2) - 1;
                    calls
                      term (n = 2)
                        executes
                          return 3 * term(1) - 1;
                              calls
                                term (n = 1)
                                  executes
                                    return 2;
```

The copy of `term` with `n = 1` now returns the value 2 to the point from which it was called and then ceases to exist. Knowing the value of `term(1)`, the value of the expression `3 * term(1) - 1` can now be completed; it gives $3 \times 2 - 1 = 5$. This value can then be returned to the point at which this version of `term` was called. This process continues until the original call computes and returns the desired value. Once again, we can illustrate the process with a diagram.



```
term (n = 1) returns 2

to term (n = 2)
  which returns 3 × 2 − 1 = 5

to term (n = 3)
  which returns 3 × 5 − 1 = 14

to term (n = 4)
  which returns 3 × 14 − 1 = 41

to the point from which it was called
```

A silly but possibly useful way of looking at this process is to imagine yourself trying to find the value of $t(4)$. Rather than doing all the calculations yourself, you phone your friend Alex. Alex knows the formula for $t(n)$, so he can see that

$t(4) = 3 \times t(3) - 1.$

Unfortunately, he doesn't know $t(3)$ so he puts you on hold while he calls his friend Kim. Kim also knows the formula so she can quickly determine that

$t(3) = 3 \times t(2) - 1.$

To help her complete her calculations, Kim must have the value of $t(2)$. To get this, she puts Alex on hold while she phones her friend Chris. Chris also knows the formula so he can see that the answer to Kim's problem is

$t(2) = 3 \times t(1) - 1$

Like the other callers before him, Chris doesn't want to do all the work so he puts Kim on hold while he phones his friend Wei Ting to find $t(1)$. Wei Ting can see from the formula that

$t(1) = 2$

Wei Ting tells Chris that the answer is 2 and then hangs up (wondering why Chris would phone to ask her such a question).
Chris can now complete his calculations:

$t(2) = 3 \times 2 - 1 = 5$

Chris passes this information back to Kim and then he too hangs up.
Kim can now determine that:

$t(3) = 3 \times 5 - 1 = 14$

Kim tells Alex her result and then she also hangs up.
Alex can now complete his calculation:

$t(4) = 3 \times 14 - 1 = 41$

He passes this back to you and hangs up, leaving you to do what you want with the knowledge that

$t(4) = 41$

If all of this calling and returning seems to you to be a waste of time, you are right! Recursion is a very useful technique but it does *not* provide the most efficient way of evaluating a function like the one that we have here. We have shown this example only because it illustrates some aspects of recursion — not because we recommend that you use it in such circumstances.

Example 2

An efficient method for calculating the value of the function of the previous example would use a loop. Here is an efficient, non-recursive method that does the job.

```
public static int term (int n)
{
  if (n < 1)
    throw new RuntimeException("Invalid parameter");
  else
  {
    int value = 2;
    for (int i = 2; i <= n; i++)
      value = 3 * value - 1;
    return value;
  }
}
```

Recursive methods in Java are not limited to evaluating terms of sequences. They can be used to evaluate any function for which we have a recursive definition.

Example 3

A more efficient form of Euclid's algorithm than the one given in the previous section replaces the successive subtractions by one mod operation (implemented by the `%` operator in Java). The revised algorithm for determining the gcd of two non-negative integers is

$$\gcd(m, n) = \begin{cases} m & \text{if } n = 0 \\ \gcd(n, m \bmod n) & \text{if } n > 0 \end{cases}$$

The algorithm can be implemented easily in Java, as shown in the following method. (The method assumes that the values of both parameters are non-negative integers.)

```
public static int gcd (int m, int n)
{
  if (n == 0)
    return m;
  else
    return gcd(n, m % n);
}
```

If we make a call to `gcd(24,18)`, we produce the following sequence of actions:

```
original call
  gcd (m = 24, n = 18)
    executes
      return gcd(18,24 % 18);
                    |
         calls      v
           gcd (m = 18, n = 6)
             executes
               return gcd(6,18 % 6);
                            |
                  calls     v
                    gcd (m = 6, n = 0)
                      executes
                        return 6;
```

Now, each copy of `gcd` returns, as shown in the next diagram.

```
gcd (m = 6, n = 0)
  returns 6
    to gcd (m = 18, n = 6)
      which returns 6
        to gcd (m = 24, n = 18)
          which returns 6
            to the point from which it was called
```

Exercises 11.3

1. Trace the execution of the method `gcd` in Example 2 to find the greatest common divisor of each pair of values.

   (a) `m` = 20 `n` = 28   (b) `m` = 991 `n` = 129