

4.16 Recursion

Self-Study Guide · About 55 minutes · With answer key

By the end you will be able to

- Explain what a recursive method is and point out its base case and its recursive call.
- Trace a recursive method call by call, keeping track of each call's own parameters and local variables.
- Work out what a recursive method prints or returns, including output that happens after the recursive call.
- Describe which starting values let a recursive method reach its base case, and what happens when it never does.
- Explain how parameter values in recursion track progress the way a loop control variable does, and match a recursive method with a loop that does the same job.

What to remember from earlier

- if and else choose between two blocks of code. ([Topic 2.3: if Statements, 2.3.3 Two-Way Selection with if-else](#))
- If the body never changes the variable in the condition, the loop never ends. ([Topic 2.7: while Loops, 2.7.4 Infinite Loops](#))
- A method call pauses the caller, runs with copies of the arguments, then returns to where it was called. ([Topic 3.5: Methods: How to Write Them, 3.5.6 Parameters Receive Copies](#))
- Parameters and local variables exist only while their method runs. ([Topic 3.8: Scope and Access, 3.8.1 Scope and Local Variables](#))
- int division drops the remainder, and % gives the remainder. ([Topic 1.3: Expressions and Output, 1.3.6 Integer Division](#))

4.16.1 A Method That Calls Itself

Every way you have repeated code so far has used a loop. To count down for a rocket launch, you would write a `while` loop like this one from Topic 2.7:

LaunchLoop.java

```

1  public class LaunchLoop
2  {
3      public static void main(String[] args)
4      {
5          int n = 3;
6          while (n > 0)
7          {
8              System.out.println(n);
9              n--;
10         }
11         System.out.println("Liftoff!");
12     }
13 }
```

```
3
2
1
Liftoff!
```

There is a second way to get repetition, and it uses nothing but methods and `if`. Think about how you would explain a countdown to a friend without saying "repeat". You might say: "To count down from 3, say 3, then count down from 2." And to count down from 2? "Say 2, then count down from 1." Each instruction hands a **smaller copy of the same job** to the next step, until the job is so small (count down from 0) that you just shout "Liftoff!" and stop.

Java lets a method work the same way. From Topic 3.5 you know a method can call another method. Nothing stops a method from calling **itself**. A **recursive method** is a method that calls itself, and the call inside its own body is the **recursive call**. Using recursive methods to repeat work is called **recursion**.

WORKED EXAMPLE A countdown with no loop

Here is the countdown again, written as a recursive method. Look for two things: the `if` that decides when to stop, and line 12, where `countdown` calls `countdown`.

LaunchRecursive.java

```
1 public class LaunchRecursive
2 {
3     public static void countdown(int n)
4     {
5         if (n == 0)
6         {
7             System.out.println("Liftoff!");
8         }
9         else
10        {
11            System.out.println(n);
12            countdown(n - 1);
13        }
14    }
15
16    public static void main(String[] args)
17    {
18        countdown(3);
19    }
20 }
```

```
3
2
1
Liftoff!
```

Main calls `countdown(3)` on line 18. Follow it step by step:

Call	n	n == 0?	What this call does
<code>countdown(3)</code>	3	false	prints 3, then calls <code>countdown(2)</code>

countdown(2)	2	false	prints 2, then calls countdown(1)
countdown(1)	1	false	prints 1, then calls countdown(0)
countdown(0)	0	true	prints Liftoff! and makes no further calls

Line 11 ran three times, yet there is no loop anywhere in the file. Each run of line 12 started the method again with a smaller number. When `n` reached 0, the `if` on line 5 chose the other branch, which does not call `countdown`, so the chain stopped. After `countdown(0)` finishes, each earlier call has nothing left to do after line 12, so they finish too, and the program ends.

The argument decides how many times the work repeats, just as the starting value of `n` decided it in the loop. Change only the calls in `main`:

WORKED EXAMPLE Changing the starting value

LaunchStarts.java

```

1  public class LaunchStarts
2  {
3      public static void countdown(int n)
4      {
5          if (n == 0)
6          {
7              System.out.println("Liftoff!");
8          }
9          else
10         {
11             System.out.println(n);
12             countdown(n - 1);
13         }
14     }
15
16     public static void main(String[] args)
17     {
18         countdown(5);
19         countdown(0);
20     }
21 }
```

```

5
4
3
2
1
Liftoff!
Liftoff!
```

`countdown(5)` makes five calls that print a number and then one more that prints `Liftoff!`. `countdown(0)` takes the stopping branch immediately: it prints `Liftoff!` and never makes a recursive call at all. A recursive method does not always call itself; the `if` decides, every single time.

KEY IDEA

A recursive method is a method that calls itself. Recursion is another form of repetition: each call does a little work and hands a smaller version of the job to the next call, until a call is small enough to finish without calling again.

On the AP exam you will be asked to read and trace recursive methods, not to write them. Every question in this guide practices reading: predicting output, finding return values, and spotting why a call never stops.

WATCH OUT

"No loop, so it runs once." Some students see `countdown(3)` in `main`, find no `while` or `for`, and predict the output is just 3. That ignores line 12. A method call runs the whole method, and line 12 is a method call, so it runs `countdown` again with 2, which runs it again with 1, and so on. When you trace a recursive method, treat every recursive call as a brand-new call that you must follow all the way through.

PAUSE AND TRY**Question 1** Tracing

Write the exact output of this program.

HopTrace.java

```
1 public class HopTrace
2 {
3     public static void hop(int n)
4     {
5         if (n < 3)
6         {
7             System.out.println("Landed at " + n);
8         }
9         else
10        {
11            System.out.println("Hop from " + n);
12            hop(n - 3);
13        }
14    }
15
16    public static void main(String[] args)
17    {
18        hop(10);
19    }
20 }
```

Question 2 Multiple choice

When `main` calls `tick(4)`, the program prints `tick` four times. Which statement explains why?

TickTock.java

```

1 public class TickTock
2 {
3     public static void tick(int count)
4     {
5         if (count > 0)
6         {
7             System.out.println("tick");
8             tick(count - 1);
9         }
10    }
11
12    public static void main(String[] args)
13    {
14        tick(4);
15    }
16 }

```

- A. Java automatically repeats any method whose parameter is greater than 0.
- B. Line 8 is a call to `tick` itself, so each call with a positive count prints once and starts another call with a count one smaller.
- C. The `if` statement on line 5 acts as a loop and repeats lines 7 and 8 while `count > 0`.
- D. `main` calls `tick` four times, once for each value of `count`.

4.16.2 The Base Case Stops the Recursion

In `LaunchRecursive.java`, what would happen if the `if` were not there and the method always called itself? A loop with a condition that never becomes false runs forever (Topic 2.7). A recursive method needs the same kind of exit, and it has a name.

Every recursive method has two kinds of paths through its body:

- A **base case** is a path that finishes **without** making a recursive call. It is the "small enough to just do it" situation, and reaching it halts the recursion. Every recursive method needs at least one.
- A **recursive case** is a path that makes at least one recursive call. Every recursive method has at least one of these too; otherwise it would not be recursive.

For recursion to stop, each recursive call must pass an argument that is **closer to the base case** than the current one. `countdown(n - 1)` moves `n` one step toward 0 every time.

Part of countdown	Line	Role
<code>if (n == 0)</code>	5	Test: is this call small enough to finish right away?
<code>println("Liftoff!")</code>	7	Base case: finishes with no recursive call
<code>println(n)</code>	11	Recursive case: this call's share of the work
<code>countdown(n - 1)</code>	12	Recursive call with a smaller argument, one step closer to 0

A base case does not have to print or return anything. In `TickTock.java` (question 2), the base case is the call where `count > 0` is false: the method skips its block and simply ends. "Do nothing and stop" is a perfectly good base case.

KEY IDEA

A recursive method needs at least one base case, which finishes without calling the method again, and at least one recursive call. Each recursive call must move the argument closer to a base case, or the calls never stop.

WORKED EXAMPLE Moving toward the base case by halving

Progress does not have to be subtracting 1. This method moves toward its base case by cutting `n` in half with integer division.

`HalveRecursive.java`

```

1 public class HalveRecursive
2 {
3     public static void halve(int n)
4     {
5         if (n == 0)
6         {
7             System.out.println("Reached 0");
8         }
9         else
10        {
11            System.out.println(n);
12            halve(n / 2);
13        }
14    }
15
16    public static void main(String[] args)
17    {
18        halve(10);
19    }
20 }
```

```

10
5
2
1
Reached 0
```

Call	n	n == 0?	Prints	Next argument
<code>halve(10)</code>	10	false	10	$10 / 2 = 5$
<code>halve(5)</code>	5	false	5	$5 / 2 = 2$
<code>halve(2)</code>	2	false	2	$2 / 2 = 1$
<code>halve(1)</code>	1	false	1	$1 / 2 = 0$
<code>halve(0)</code>	0	true	Reached 0	none

Integer division drops the remainder (Topic 1.3), so $5 / 2$ is 2 and $1 / 2$ is 0. Because every positive number shrinks under $/ 2$, the argument always reaches 0 for any positive starting value.

WATCH OUT

A recursive method with no base case. Remove the `if` from `countdown`, and every call makes another call:

NoBaseCase.java

```
1 public class NoBaseCase
2 {
3     public static void countdown(int n)
4     {
5         System.out.println(n);
6         countdown(n - 1);
7     }
8
9     public static void main(String[] args)
10    {
11        countdown(3);
12    }
13 }
```

The program prints 3, 2, 1, 0, -1, -2, and keeps going into negative numbers. It does not run forever like an infinite loop, though. It crashes with an error like this (the exact wording and the long list of lines after it vary between Java versions):

```
Exception in thread "main" java.lang.StackOverflowError
```

Here is why. A call that has made a recursive call is not finished; it is **waiting** for that call to return, and Java must remember every waiting call along with its variables. With no base case, the waiting calls pile up until Java runs out of the memory set aside for them. That memory is called the **call stack**, and running out of it is reported as a **StackOverflowError**. When you see this error, look for a missing base case or for recursive calls that never get closer to one. (Even a correct recursive method can overflow if it needs an enormous number of calls before reaching its base case. AP questions use small starting values, and so does this guide.)

PAUSE AND TRY

Question 3 Debugging

`laps(6)` prints 6 to go, 4 to go, 2 to go, Finished. But `laps(5)` prints 5 to go, 3 to go, 1 to go, -1 to go, and keeps going until the program crashes with `StackOverflowError`. Find the line that causes the problem, explain why, and write a corrected line.

LapCounter.java

```
1 public class LapCounter
2 {
3     public static void laps(int remaining)
4     {
5         if (remaining == 0)
6         {
7             System.out.println("Finished");
8         }
9         else
10        {
11            System.out.println(remaining + " to go");
12            laps(remaining - 2);
13        }
14    }
15
16    public static void main(String[] args)
17    {
18        laps(6);
19        laps(5);
20    }
21 }
```

Question 4 Multiple choice

For which starting values of `step` does `climb(step)` print `Top` and finish without an error? (As in the rest of this guide, assume starting values are small.)

ClimbCheck.java

```
1 public class ClimbCheck
2 {
3     public static void climb(int step)
4     {
5         if (step == 10)
6         {
7             System.out.println("Top");
8         }
9         else
10        {
11            climb(step + 1);
12        }
13    }
14
15    public static void main(String[] args)
16    {
17        climb(4);
18        climb(10);
19        climb(-3);
20        climb(12);
21    }
22 }
```

A. Only `step == 10`

- B. Every value of `step`
- C. Every value of `step` that is 10 or less
- D. Every value of `step` that is 10 or more

4.16.3 Each Call Has Its Own Variables

So far every recursive call has been the last thing its method did. What if there is code **after** the recursive call? To trace it you need to answer a question you may already be wondering about: when `countdown(3)` calls `countdown(2)`, the new call sets `n` to 2. Did the first call's `n` just get overwritten?

It did not. Recall from Topic 3.5 that every call to a method gets **fresh** parameters, filled with copies of the arguments, and from Topic 3.8 that parameters and local variables exist only while their call is running. Those rules do not change when the method calling is the same method. Each recursive call gets its own `n` and its own copy of every local variable, completely separate from the `n` of the call that is waiting for it.

A good picture is a stack of sticky notes. Starting a call puts a new note on top, with that call's variables written on it. Only the top note is being worked on; the notes underneath are calls that are paused at their recursive call. When the top call finishes, its note is thrown away and the call just below picks up **exactly where it paused**, with its own variables untouched. This stack of waiting calls is the call stack from 4.16.2. (Unlike sticky notes, you never write or erase the notes yourself; Java does it on every call and return.)

KEY IDEA

Every recursive call gets a private copy of its parameters and of every variable declared inside the method. When a call finishes, the call that made it resumes right after the recursive call, with its own values unchanged.

WORKED EXAMPLE Printing after the recursive call

This method is almost `countdown`, but line 12 prints **after** the recursive call on line 11. Predict the output before reading on.

CountUp.java

```

1 public class CountUp
2 {
3     public static void countUp(int n)
4     {
5         if (n == 0)
6         {
7             System.out.println("Start");
8         }
9         else
10        {
11            countUp(n - 1);
12            System.out.println(n);
13        }
14    }
15
16    public static void main(String[] args)
17    {
18        countUp(3);
19    }
20 }

```

Start

```

1
2
3

```

The numbers come out counting **up**. Here is the whole run, indented one step for each call deeper:

```

countUp(3)      n = 3, pauses at line 11
  countUp(2)    n = 2, pauses at line 11
    countUp(1)  n = 1, pauses at line 11
      countUp(0) n = 0, base case: prints Start
    countUp(1) resumes at line 12: prints its n, 1
  countUp(2) resumes at line 12: prints its n, 2
countUp(3) resumes at line 12: prints its n, 3

```

At the moment `countUp(0)` is running, the call stack holds four calls, each with its own `n`:

Position	Call	Its n	Waiting to do
top (running)	countUp(0)	0	print Start
waiting	countUp(1)	1	line 12: print 1
waiting	countUp(2)	2	line 12: print 2
bottom (waiting)	countUp(3)	3	line 12: print 3

No call can reach line 12 until the call it made has completely finished. So the first `println(n)` to run belongs to the deepest call, and the last belongs to the very first call. Work done after a recursive call happens in the reverse order of the calls.

Local variables declared inside the method behave the same way as parameters. In the next example, each call computes its own `doubled` before the recursive call and uses it again afterward.

WORKED EXAMPLE A local variable in every call

DownAndUp.java

```

1  public class DownAndUp
2  {
3      public static void echo(int n)
4      {
5          int doubled = n * 2;
6          if (n == 0)
7          {
8              System.out.println("Bottom");
9          }
10         else
11         {
12             System.out.println("Down " + doubled);
13             echo(n - 1);
14             System.out.println("Up " + doubled);
15         }
16     }
17
18     public static void main(String[] args)
19     {
20         echo(2);
21     }
22 }
```

```

Down 4
Down 2
Bottom
Up 2
Up 4
```

Step	Running	Its n	Its doubled	Prints
1	echo(2)	2	4	Down 4
2	echo(1)	1	2	Down 2
3	echo(0)	0	0	Bottom
4	echo(1) resumes	1	2	Up 2
5	echo(2) resumes	2	4	Up 4

Line 13 splits each call in two: line 12 runs on the way down, and line 14 runs on the way back up. When `echo(1)` resumes in step 4, its `doubled` is still 2, even though `echo(0)` set a `doubled` of its own to 0. They were never the same variable.

WATCH OUT

"The base case ends the program." For `CountUp.java`, many students predict only `Start`, reasoning that the base case "stops the recursion" so nothing else happens. Others predict

3, 2, 1, Start, reading the prints as if they came before the call. Both forget the waiting calls. Reaching the base case stops new calls from being made, but every paused call still has to resume and finish the lines after its recursive call. When code follows a recursive call, trace the way back up as carefully as the way down.

PAUSE AND TRY

Question 5 Tracing

Line 7 changes `n` before the recursive call. Write the exact output.

ShrinkTrace.java

```

1  public class ShrinkTrace
2  {
3      public static void shrink(int n)
4      {
5          if (n > 0)
6          {
7              n = n - 1;
8              shrink(n);
9              System.out.println(n);
10         }
11     }
12
13     public static void main(String[] args)
14     {
15         shrink(3);
16     }
17 }
```

Question 6 Short answer

In `CountUp.java`, `main` calls `countUp(3)`, so 3 is the first argument used. Explain why 3 is the **last** number printed.

4.16.4 Returning Values Through the Calls

Printing is one way a recursive method can produce results. More often, a recursive method **returns** a value. Suppose you want the total of $1 + 2 + \dots + n$. You could add the numbers with a loop. But notice that the total up to 4 is just 4 plus the total up to 3, and the total up to 3 is 3 plus the total up to 2, and so on, down to the total up to 1, which is simply 1. Each total is described using a smaller total: that is a recursive description.

From Topic 3.5, a non-`void` method's call is replaced by the value it returns, and the caller continues the expression with that value. In a recursive method, the caller is just an earlier call to the same method, waiting in the middle of an expression for the answer.

WORKED EXAMPLE Adding 1 through n

SumTo.java

```

1 public class SumTo
2 {
3     public static int sumTo(int n)
4     {
5         if (n == 1)
6         {
7             return 1;
8         }
9         return n + sumTo(n - 1);
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(sumTo(4));
15    }
16 }

```

10

Line 7 is the base case: the total up to 1 is 1. Line 9 is the recursive case: it cannot finish $n + \text{sumTo}(n - 1)$ until $\text{sumTo}(n - 1)$ returns a number. A good way to trace this is to write each call, then replace it with its returned value on the way back:

```

sumTo(4) = 4 + sumTo(3)
           sumTo(3) = 3 + sumTo(2)
                       sumTo(2) = 2 + sumTo(1)
                               sumTo(1) = 1      base case
                                   sumTo(2) = 2 + 1 = 3
                                       sumTo(3) = 3 + 3 = 6
                                           sumTo(4) = 4 + 6 = 10

```

Call	Its n	Waits for	Value it gets back	Returns
sumTo(4)	4	sumTo(3)	6	$4 + 6 = 10$
sumTo(3)	3	sumTo(2)	3	$3 + 3 = 6$
sumTo(2)	2	sumTo(1)	1	$2 + 1 = 3$
sumTo(1)	1	nothing		1

Read the table from the top down for the calls being made, then from the bottom up for the answers coming back. No addition happens at all until the base case returns 1.

A method can have more than one base case. This one answers a yes-or-no question by stepping down by 3, and it needs one base case for "landed exactly on 0" and another for "went past 0".

WORKED EXAMPLE Two base cases

MultipleOfThree.java

```

1 public class MultipleOfThree
2 {
3     public static boolean isMultipleOfThree(int n)
4     {
5         if (n == 0)
6         {
7             return true;
8         }
9         if (n < 0)
10        {
11            return false;
12        }
13        return isMultipleOfThree(n - 3);
14    }
15
16    public static void main(String[] args)
17    {
18        System.out.println(isMultipleOfThree(9));
19        System.out.println(isMultipleOfThree(7));
20    }
21 }

```

```

true
false

```

Call	Base case?	Returns
isMultipleOfThree(9)	no	the result of the call with 6
isMultipleOfThree(6)	no	the result of the call with 3
isMultipleOfThree(3)	no	the result of the call with 0
isMultipleOfThree(0)	yes, line 5	true

So `true` is handed back unchanged through every call. For 7, the calls go 7, 4, 1, then -2, which hits the second base case on line 9 and returns `false`. Without that second base case, 7 would behave like `laps(5)` in question 3 and skip past 0 forever. The method expects a starting value of 0 or more; for a negative multiple of 3 such as -6, it would return `false` right away.

WATCH OUT

Ignoring the value a recursive call returns. A student writes the recursive call on its own line and then returns `n`:

SumToIgnored.java

```
1 public class SumToIgnored
2 {
3     public static int sumTo(int n)
4     {
5         if (n == 1)
6         {
7             return 1;
8         }
9         sumTo(n - 1);
10        return n;
11    }
12
13    public static void main(String[] args)
14    {
15        System.out.println(sumTo(4));
16    }
17 }
```

4

It compiles, and line 9 really does run `sumTo(3)`, which computes 6. But calling a method as a statement throws its return value away (Topic 3.5), so the 6 is lost and line 10 returns only `n`, which is 4. The returned value must be **used** in the expression: `return n + sumTo(n - 1);`.

PAUSE AND TRY

Question 7 Tracing

What does this program print? Show the value each call returns.

ProductOdd.java

```
1 public class ProductOdd
2 {
3     public static int product(int n)
4     {
5         if (n <= 1)
6         {
7             return 1;
8         }
9         return n * product(n - 2);
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(product(7));
15    }
16 }
```

Question 8 Multiple choice

Using `sumTo` from `SumTo.java`, what happens when a program calls `sumTo(0)`?

- A. It returns 0.
- B. It returns 1, because the base case returns 1.
- C. It crashes with `StackOverflowError`, because the argument never equals 1.
- D. It does not compile, because 0 is not allowed as an argument.

4.16.5 Parameters Track the Progress

When you trace a loop, the loop control variable tells you where you are: in

`for (int i = 3; i <= 6; i++)`, seeing `i == 5` means the loop is on its third pass and has two values left (Topics 2.7 and 2.8). A recursive method has no loop control variable. So what plays that role?

The parameters do. Each call receives its own parameter values, and those values describe how far the process has come. In `countdown`, a call with `n == 1` is one step from the base case, just as `n == 1` in `LaunchLoop.java` is one pass from the end of the loop. The difference is where the value lives: a loop **changes** one variable over and over, while recursion gives each call a **new** parameter that holds the next value.

The match is easiest to see when a recursive method carries everything a loop would track, including a running total, in its parameters.

WORKED EXAMPLE The same sum, as a loop and as recursion

Both methods add the whole numbers from 3 through 6.

AddRange.java

```

1 public class AddRange
2 {
3     public static int addLoop(int start, int stop)
4     {
5         int total = 0;
6         for (int i = start; i <= stop; i++)
7         {
8             total = total + i;
9         }
10        return total;
11    }
12
13    public static int addRecursive(int i, int stop, int total)
14    {
15        if (i > stop)
16        {
17            return total;
18        }
19        return addRecursive(i + 1, stop, total + i);
20    }
21
22    public static void main(String[] args)
23    {
24        System.out.println(addLoop(3, 6));
25        System.out.println(addRecursive(3, 6, 0));
26    }
27 }

```

```

18
18

```

In `addLoop`, the variables `i` and `total` change on every pass. In `addRecursive`, there are no variables that change; instead each call is passed the next `i` and the next `total` on line 19. Put the passes of the loop next to the calls:

Loop pass	i, total at start of pass	Recursive call	Its i, total
1	3, 0	<code>addRecursive(3, 6, 0)</code>	3, 0
2	4, 3	<code>addRecursive(4, 6, 3)</code>	4, 3
3	5, 7	<code>addRecursive(5, 6, 7)</code>	5, 7
4	6, 12	<code>addRecursive(6, 6, 12)</code>	6, 12
loop ends	7, 18 (7 <= 6 is false)	<code>addRecursive(7, 6, 18)</code>	base case: returns 18

The rows match exactly. The loop's condition `i <= 6` failing is the same moment as the base case `i > stop` becoming true. The loop's update `i++` matches the argument `i + 1`, and `total = total + i` matches the argument `total + i`. The 18 returned by the last call is passed back unchanged through every waiting call, because line 19 returns the recursive call's value directly.

Parameters can track progress in any quantity, not just a counter. Here the parameter `balance` records how far a savings account has grown.

WORKED EXAMPLE Counting years until a goal

Each year the balance grows by one tenth (using integer division). The method returns how many years it takes to reach the goal.

YearsToGoal.java

```

1 public class YearsToGoal
2 {
3     public static int yearsToGoal(int balance, int goal)
4     {
5         if (balance >= goal)
6         {
7             return 0;
8         }
9         return 1 + yearsToGoal(balance + balance / 10, goal);
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(yearsToGoal(1000, 1300));
15        System.out.println(yearsToGoal(1500, 1300));
16    }
17 }
```

3
0

Call	balance >= goal?	Returns
yearsToGoal(1000, 1300)	no	1 + (next call) = 3
yearsToGoal(1100, 1300)	no	1 + (next call) = 2
yearsToGoal(1210, 1300)	no	1 + (next call) = 1
yearsToGoal(1331, 1300)	yes	0

Reading down the `balance` column is like watching a loop variable change: 1000, 1100, 1210, 1331. Each `1 +` on line 9 counts one year on the way back up. The second call in `main`, `yearsToGoal(1500, 1300)`, starts at the goal already, so it hits the base case and returns 0 without any recursive calls.

KEY IDEA

The parameters of each call show how far a recursive process has come, just as the loop control variable shows how far a loop has come. Reading the parameter values call by call tells you how close the process is to its base case.

WATCH OUT

Forgetting to update the progress parameter. In a loop, forgetting `i++` gives an infinite loop (Topic 2.7). The recursive version of that mistake is passing the progress parameter along unchanged:

AddRangeStuck.java

```
1 public class AddRangeStuck
2 {
3     public static int addRecursive(int i, int stop, int total)
4     {
5         if (i > stop)
6         {
7             return total;
8         }
9         return addRecursive(i, stop, total + i);
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(addRecursive(3, 6, 0));
15    }
16 }
```

Line 9 passes `i` instead of `i + 1`. Every call receives `i == 3`, so `i > stop` never becomes true. Nothing prints; the program crashes with `java.lang.StackOverflowError`. The loop version of this mistake would run forever, but the recursive version stops with an error, because each waiting call uses up a little more of the call stack. When you check whether a recursive method ends, look at the arguments in the recursive call: at least one must move toward the base case.

PAUSE AND TRY

Question 9 Tracing

`halvings` counts how many times a number can be halved (with integer division) before it reaches 1 or less. Trace `halvings(40, 0)` in a table showing each call's parameters, and write the output.

Halvings.java

```
1 public class Halvings
2 {
3     public static int halvings(int n, int count)
4     {
5         if (n <= 1)
6         {
7             return count;
8         }
9         return halvings(n / 2, count + 1);
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(halvings(40, 0));
15    }
16 }
```

Question 10 Multiple choice

While tracing `addRecursive(3, 6, 0)` from `AddRange.java`, you reach the call `addRecursive(5, 6, 7)`. What does the parameter value 7 tell you?

- A. Seven recursive calls have been made so far.
- B. The numbers already added, 3 and 4, total 7.
- C. The final answer will be 7.
- D. The number 7 will be added next.

4.16.6 Recursion and Loops Can Replace Each Other

You have now seen several jobs done both ways: the countdown in 4.16.1 and the range sum in 4.16.5. That was no coincidence. Any recursive solution can be rewritten as an **iterative** one (one that repeats with a loop), and any iterative solution can be rewritten recursively. Neither tool can solve a problem the other cannot. Programmers choose whichever version is easier to read and reason about for the problem at hand.

To match the two versions, line up their parts:

In the recursive version	In the loop version
The base case	The loop condition becoming false
The changed argument in the recursive call	The update to the loop control variable
The work combined with the call's result	The work done in the loop body
The value the base case returns	The starting value of the result variable

KEY IDEA

Any recursive solution can be written with a loop instead, and any loop solution can be written with recursion. The two approaches can always produce the same results.

WORKED EXAMPLE Adding digits both ways

`digitSumRecursive` adds the digits of a nonnegative number. `n % 10` is the last digit and `n / 10` is the number with its last digit removed (Topic 1.3). `digitSumLoop` does the same job with a `while` loop.

DigitSum.java

```

1  public class DigitSum
2  {
3      public static int digitSumRecursive(int n)
4      {
5          if (n < 10)
6          {
7              return n;
8          }
9          return n % 10 + digitSumRecursive(n / 10);
10     }
11
12     public static int digitSumLoop(int n)
13     {
14         int sum = 0;
15         while (n > 0)
16         {
17             sum = sum + n % 10;
18             n = n / 10;
19         }
20         return sum;
21     }
22
23     public static void main(String[] args)
24     {
25         System.out.println(digitSumRecursive(4721));
26         System.out.println(digitSumLoop(4721));
27         System.out.println(digitSumRecursive(7));
28         System.out.println(digitSumLoop(7));
29     }
30 }

```

```

14
14
7
7

```

Trace both with 4721:

Recursive call	Last digit	Loop pass	n before	sum after
digitSumRecursive(4721)	1	1	4721	1
digitSumRecursive(472)	2	2	472	3
digitSumRecursive(47)	7	3	47	10
digitSumRecursive(4)	base: 4	4	4	14

The recursive version returns 4 from its base case, then adds on the way back: $7 + 4 = 11$, then $2 + 11 = 13$, then $1 + 13 = 14$. The loop adds from the other end, $1 + 2 + 7 + 4$, and reaches the same 14 once n becomes 0. The parameter n in each recursive call holds the same values as the variable n at the start of each loop pass, which is the idea from 4.16.5 again.

The translation works in the other direction too. Start from a loop you might write yourself and look at its recursive twin.

WORKED EXAMPLE Starting from a loop

PowerBothWays.java

```

1  public class PowerBothWays
2  {
3      public static int powerLoop(int base, int exponent)
4      {
5          int result = 1;
6          for (int count = exponent; count > 0; count--)
7          {
8              result = result * base;
9          }
10         return result;
11     }
12
13     public static int powerRecursive(int base, int exponent)
14     {
15         if (exponent == 0)
16         {
17             return 1;
18         }
19         return base * powerRecursive(base, exponent - 1);
20     }
21
22     public static void main(String[] args)
23     {
24         System.out.println(powerLoop(2, 5));
25         System.out.println(powerRecursive(2, 5));
26         System.out.println(powerLoop(3, 4));
27         System.out.println(powerRecursive(3, 4));
28     }
29 }

```

```

32
32
81
81

```

`powerLoop` starts `result` at 1 and multiplies by `base` once per pass while `count` goes down to 0. `powerRecursive` says the same thing as a rule: anything to the power 0 is 1 (the base case on line 17, matching `result = 1` on line 5), and `base` to the power `exponent` is `base` times `base` to the power `exponent - 1` (line 19, matching `count--` and the multiplication on line 8). For `powerRecursive(2, 5)` the calls use exponents 5, 4, 3, 2, 1, 0, and the multiplications on the way back give 1, 2, 4, 8, 16, 32.

The two versions differ in one practical way you have already seen. The recursive version keeps a waiting call on the call stack for every step, so a recursive process with a very large number of steps, or with no base case, can end in `StackOverflowError`. A loop uses the same few variables on every pass. The results of a correct recursive version and a correct loop version are the same.

WATCH OUT

Copying the base case into the loop condition. A student turns `digitSumRecursive` into a loop by reusing its test, `n < 10`, as the place to stop:

`DigitSumLoopBug.java`

```

1 public class DigitSumLoopBug
2 {
3     public static int digitSumLoop(int n)
4     {
5         int sum = 0;
6         while (n >= 10)
7         {
8             sum = sum + n % 10;
9             n = n / 10;
10        }
11        return sum;
12    }
13
14    public static void main(String[] args)
15    {
16        System.out.println(digitSumLoop(4721));
17    }
18 }
```

10

The loop stops as soon as `n` is a single digit, so the passes add 1, 2, and 7, and the loop ends with `n` equal to 4, which is never added. In the recursive version, the base case did not just stop: it **returned** `n`, adding that last digit. A loop has no such return, so it must keep going until `n` is 0, as in `while (n > 0)`. When you translate, make sure the work the base case does still happens.

PAUSE AND TRY**Question 11** Multiple choice

Which loop computes the same value as `sumEveryOther(n)` for every `n` greater than or equal to 0?

```

1 public static int sumEveryOther(int n)
2 {
3     if (n <= 0)
4     {
5         return 0;
6     }
7     return n + sumEveryOther(n - 2);
8 }
```

Each choice is the body of a method that ends with `return total;`.

A. `int total = 0; then for (int k = n; k > 0; k -= 2) { total += k; }`

- B. `int total = 0; then for (int k = 1; k <= n; k += 2) { total += k; }`
- C. `int total = 0; then for (int k = n; k >= 0; k--) { total += k; }`
- D. `int total = 0; int k = n; then while (k > 0) { k -= 2; total += k; }`

Question 12 Short answer

A classmate says: "Some problems can only be solved with recursion, and others can only be solved with loops." Is the classmate right? Explain using an example from this section.

Putting it together

A staircase has `n` steps, and you climb by taking either 1 step or 2 steps at a time. How many different ways can you reach the top? For 3 steps there are three ways: 1+1+1, 1+2, and 2+1. Here is a recursive way to think about it: your **first move** is either 1 step, leaving `n - 1` steps to climb, or 2 steps, leaving `n - 2`. So the number of ways for `n` steps is the ways for `n - 1` plus the ways for `n - 2`. A staircase of 0 or 1 steps has exactly 1 way.

This method uses every idea in the chapter, plus one new twist: the recursive case makes **two** recursive calls. Before reading the trace, predict `ways(4)`.

WORKED EXAMPLE Ways to climb the stairs

StairWays.java

```

1  public class StairWays
2  {
3      public static int ways(int n)
4      {
5          if (n <= 1)
6          {
7              return 1;
8          }
9          return ways(n - 1) + ways(n - 2);
10     }
11
12     public static int waysLoop(int n)
13     {
14         int before = 1;
15         int current = 1;
16         for (int step = 2; step <= n; step++)
17         {
18             int next = current + before;
19             before = current;
20             current = next;
21         }
22         return current;
23     }
24
25     public static void main(String[] args)
26     {
27         System.out.println(ways(4));
28         System.out.println(waysLoop(4));

```

```

29         System.out.println(ways(6));
30         System.out.println(waysLoop(6));
31     }
32 }

```

```

5
5
13
13

```

`ways` has one base case (line 7, for `n` of 0 or 1) and a recursive case on line 9 with two recursive calls. Java evaluates `ways(n - 1) + ways(n - 2)` left to right: it finishes the **entire** first call, with all the calls that one makes, before it starts the second. Each call has its own `n`, and each returns its value to the call that made it. Indented one level per call deeper:

```

ways(4)
  ways(3)
    ways(2)
      ways(1) returns 1      base case
      ways(0) returns 1      base case
      ways(2) returns 1 + 1 = 2
      ways(1) returns 1      base case
    ways(3) returns 2 + 1 = 3
  ways(2)
    ways(1) returns 1      base case
    ways(0) returns 1      base case
    ways(2) returns 1 + 1 = 2
  ways(4) returns 3 + 2 = 5

```

Things to notice:

- **Base case and progress.** Both recursive calls pass a smaller `n`, and from any `n` of 0 or more, subtracting 1 or 2 always lands on 0 or 1. Every path down the trace ends at a base case.
- **Each call's own variables.** When `ways(3)` gets 2 back from its first call, it still knows its own `n` is 3, so its second call is `ways(1)`.
- **Return values.** Nothing is added until base cases return; the sums happen on the way back up.
- **Iteration does the same job.** `waysLoop` gets the same answers with a `for` loop. Its variables `before` and `current` hold the answers for the two previous staircase sizes, and `step` tracks progress the way `n` does in the recursive version. Notice that the recursive trace computes `ways(2)` twice; the loop computes each size once.

Vocabulary

Term	Definition	Example
recursive method	A method that calls itself	countdown in <code>LaunchRecursive</code>

recursive call	A call to a method from inside that same method's body	<code>countdown(n - 1);</code>
recursion	Repeating work by having a method call itself on a smaller version of the job	counting down without a loop
base case	A path through a recursive method that finishes without a recursive call, halting the recursion	<code>if (n == 0) → print Liftoff!</code>
recursive case	A path through a recursive method that makes at least one recursive call	<code>return n + sumTo(n - 1);</code>
call stack	The calls that have started but not finished, newest on top; each keeps its own variables	<code>countUp(3)</code> under <code>countUp(2)</code>
<code>StackOverflowError</code>	The runtime error when waiting calls use up the call stack, usually from a missing or unreachable base case	<code>sumTo(0)</code>
iterative	Describes a solution that repeats with a loop rather than with recursion	<code>digitSumLoop</code>

Chapter summary

- A recursive method calls itself. It needs at least one base case, which finishes without calling again, and at least one recursive call. Recursion is another way to repeat work.
- Each recursive call must move its arguments closer to a base case. If the base case is missing or can be skipped over, the waiting calls pile up and the program crashes with `StackOverflowError`.
- Every call has its own parameters and local variables. A call that makes a recursive call waits, then resumes right after it with its own values unchanged, so code after a recursive call runs in reverse order of the calls.
- A recursive method that returns a value builds its answer on the way back: each call combines its own work with the value its recursive call returns.
- Parameter values show the progress of a recursive process, just as the loop control variable shows the progress of a loop.
- Any recursive solution can be rewritten with a loop, and any loop solution can be rewritten with recursion.

Review questions

Question 13 Multiple choice

When `main` calls `fold(7)`, how many calls to `fold` happen in total (counting the first one), and what value is returned to `main`?

FoldCalls.java

```
1 public class FoldCalls
2 {
3     public static int fold(int n)
4     {
5         if (n <= 2)
6         {
7             return n;
8         }
9         return fold(n - 2) + 1;
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(fold(7));
15    }
16 }
```

- A. 4 calls; returns 4
- B. 3 calls; returns 4
- C. 4 calls; returns 7
- D. 3 calls; returns 3

Question 14 Tracing

Write the exact output of this program.

PatternTrace.java

```
1 public class PatternTrace
2 {
3     public static void pattern(int n)
4     {
5         if (n <= 0)
6         {
7             System.out.println("|");
8         }
9         else
10        {
11            System.out.println("<" + n);
12            pattern(n - 2);
13            System.out.println(n + ">");
14        }
15    }
16
17    public static void main(String[] args)
18    {
19        pattern(5);
20    }
21 }
```

Question 15 Tracing

What does this program print? Show what each call returns.

BinaryConvert.java

```
1 public class BinaryConvert
2 {
3     public static int convert(int n)
4     {
5         if (n < 2)
6         {
7             return n;
8         }
9         return convert(n / 2) * 10 + n % 2;
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(convert(11));
15    }
16 }
```

Question 16 Multiple choice

Which describes exactly the starting values of `n` for which `drain(n)` prints `Empty` and finishes without an error? Assume starting values are small.

DrainCheck.java

```
1 public class DrainCheck
2 {
3     public static void drain(int n)
4     {
5         if (n == 0)
6         {
7             System.out.println("Empty");
8         }
9         else
10        {
11            drain(n - 4);
12        }
13    }
14
15    public static void main(String[] args)
16    {
17        drain(8);
18        drain(0);
19        drain(6);
20    }
21 }
```

- A. All values of `n` that are 0 or more
- B. All values of `n` that are 0 or more and divisible by 4
- C. All values of `n` that are divisible by 4, including negative ones
- D. All values of `n` that are 4 or more

Question 17 Debugging

`factorial(4)` is supposed to return $4 \times 3 \times 2 \times 1 = 24$, but the program prints `0`. Find the error, explain it, and write the corrected line.

FactorialBug.java

```
1 public class FactorialBug
2 {
3     public static int factorial(int n)
4     {
5         if (n == 1)
6         {
7             return 0;
8         }
9         return n * factorial(n - 1);
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(factorial(4));
15    }
16 }
```

Question 18 Tracing

Trace `reverseDigits(504, 0)` in a table showing each call's parameters, and write the output.

ReverseDigits.java

```
1 public class ReverseDigits
2 {
3     public static int reverseDigits(int n, int built)
4     {
5         if (n == 0)
6         {
7             return built;
8         }
9         return reverseDigits(n / 10, built * 10 + n % 10);
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(reverseDigits(504, 0));
15    }
16 }
```

Question 19 Multiple choice

What value does `count(4)` return?

CountCalls.java

```
1 public class CountCalls
2 {
3     public static int count(int n)
4     {
5         if (n <= 1)
6         {
7             return 1;
8         }
9         return count(n - 1) + count(n - 2) + 1;
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(count(4));
15    }
16 }
```

- A. 5
- B. 7
- C. 8
- D. 9

Reflection

- ☐ I can explain what makes a method recursive and why recursion is a form of repetition. Reread: 4.16.1 A Method That Calls Itself
- ☐ I can point out the base case and the recursive call, and decide which starting values reach the base case. Reread: 4.16.2 The Base Case Stops the Recursion
- ☐ I can explain what causes a StackOverflowError in a recursive method. Reread: 4.16.2 The Base Case Stops the Recursion
- ☐ I can trace output that comes before and after a recursive call, keeping each call's own variables separate. Reread: 4.16.3 Each Call Has Its Own Variables
- ☐ I can find the value a recursive method returns by working back up from the base case. Reread: 4.16.4 Returning Values Through the Calls
- ☐ I can use parameter values to describe how far a recursive process has progressed. Reread: 4.16.5 Parameters Track the Progress
- ☐ I can match a recursive method with a loop that produces the same results. Reread: 4.16.6 Recursion and Loops Can Replace Each Other

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
Hop from 10
Hop from 7
Hop from 4
Landed at 1
```

Call	n	n < 3?	Prints	Next call
hop(10)	10	false	Hop from 10	hop(7)
hop(7)	7	false	Hop from 7	hop(4)
hop(4)	4	false	Hop from 4	hop(1)
hop(1)	1	true	Landed at 1	none

The argument drops by 3 each time. The chain stops at 1 because 1 is less than 3; it never needs to land exactly on 0.

Reread 4.16.1 *A Method That Calls Itself* if your output differed.

Question 2

B. `tick` is recursive: line 8 calls `tick(count - 1)`. Calls with counts 4, 3, 2, and 1 each print `tick` and make the next call. The call `tick(0)` finds the condition false, does nothing, and the chain stops.

- **A** is wrong: Java never repeats a method on its own. Repetition here comes only from the call on line 8.
- **C** is wrong: an `if` runs its block at most once per call. The repetition comes from the method being called again, not from the `if`.
- **D** is wrong: `main` makes exactly one call, `tick(4)`. The other three calls are made by `tick` itself.

Reread 4.16.1 *A Method That Calls Itself* if you chose differently.

Question 3

Line 5 is the problem. The base case only catches `remaining` equal to exactly 0. Starting from 5 and subtracting 2 gives 5, 3, 1, -1, -3, ... : odd numbers that skip right over 0. The base case is never reached, so the waiting calls pile up until the call stack runs out.

Corrected line 5:

```
1         if (remaining <= 0)
```

Now the call with -1 is a base case. With that change, `laps(5)` prints 5 to go, 3 to go, 1 to go, Finished, and `laps(6)` still works as before. Java gave no compile error for the original because the code is legal; the problem appears only when the program runs with an odd starting value.

Reread 4.16.2 *The Base Case Stops the Recursion* if you missed it.

Question 4

C. Each recursive call adds 1, so the argument climbs upward. From any value at or below 10, it eventually equals 10 and the base case prints `Top`. From a value above 10, adding 1 moves it **away** from 10 forever, so the calls pile up and the program crashes with `StackOverflowError`. That is exactly what the program shows: `climb(4)`, `climb(10)`, and `climb(-3)` each print `Top`, and then `climb(12)` crashes.

- **A** is wrong: 10 works without any recursive call, but smaller values work too; they just make some recursive calls first.
- **B** is wrong: starting above 10 never reaches the base case.
- **D** is wrong: this has the direction backwards. The argument only gets larger, so it can reach 10 only from below.

Reread 4.16.2 *The Base Case Stops the Recursion* if you chose differently.

Question 5

```
0
1
2
```

Step	Running	Its <code>n</code> after line 7	Action
1	<code>shrink(3)</code>	2	calls <code>shrink(2)</code> and waits
2	<code>shrink(2)</code>	1	calls <code>shrink(1)</code> and waits
3	<code>shrink(1)</code>	0	calls <code>shrink(0)</code> and waits
4	<code>shrink(0)</code>	(line 7 skipped)	<code>0 > 0</code> is false: base case, ends
5	<code>shrink(1)</code> resumes	0	line 9 prints 0
6	<code>shrink(2)</code> resumes	1	line 9 prints 1
7	<code>shrink(3)</code> resumes	2	line 9 prints 2

Each call subtracted 1 from **its own** `n`, and later calls never touched it. The first call's `n` started at 3, became 2 on line 7, and was still 2 when that call printed last.

Reread 4.16.3 *Each Call Has Its Own Variables* if your output differed.

Question 6

A model answer: The call `countUp(3)` reaches line 11 and calls `countUp(2)` before it prints anything. It pauses there and cannot reach line 12 until `countUp(2)` is completely finished, and `countUp(2)` in turn waits for `countUp(1)`, which waits for `countUp(0)`. So the printing happens only as the calls finish, deepest first. `countUp(3)` was the first to start and is the last to finish, so its `println(n)` runs last and prints its own `n`, which is still 3.

A correct answer must mention:

- the print comes after the recursive call, so each call waits before printing;
- calls finish in the reverse order they started (the deepest one first);
- the first call's own `n` is still 3 when it finally prints.

Reread 4.16.3 *Each Call Has Its Own Variables* if your explanation missed one of these.

Question 7

105

```

product(7) = 7 * product(5)
product(5) = 5 * product(3)
product(3) = 3 * product(1)
product(1) = 1           base case (1 <= 1)
product(3) = 3 * 1 = 3
product(5) = 5 * 3 = 15
product(7) = 7 * 15 = 105

```

The calls go down by 2 from 7 to 1, which is a base case. Then the multiplications happen on the way back: 3, then 15, then 105.

Reread *4.16.4 Returning Values Through the Calls* if your answer differed.

Question 8

C. The only base case is `n == 1`. Starting at 0, the recursive calls use `-1`, `-2`, `-3`, ..., moving away from 1, so the base case is never reached and the waiting calls overflow the call stack. `sumTo` works only when its argument is 1 or more; that is a condition the caller must meet before calling it.

- **A** is wrong: the method has no line that returns 0. You might expect "the sum up to 0 is 0", but the code does not say that.
- **B** is wrong: the base case returns 1 only when `n` is 1, and starting from 0 the argument never becomes 1.
- **D** is wrong: the parameter is an `int`, and 0 is an `int`. Java checks types when compiling, not whether the recursion will ever stop.

Reread *4.16.4 Returning Values Through the Calls* and *4.16.2 The Base Case Stops the Recursion* if you chose differently.

Question 9

5

Call	<code>n <= 1?</code>	Next call
<code>halvings(40, 0)</code>	no	<code>halvings(20, 1)</code>
<code>halvings(20, 1)</code>	no	<code>halvings(10, 2)</code>
<code>halvings(10, 2)</code>	no	<code>halvings(5, 3)</code>
<code>halvings(5, 3)</code>	no	<code>halvings(2, 4)</code>
<code>halvings(2, 4)</code>	no	<code>halvings(1, 5)</code>
<code>halvings(1, 5)</code>	yes	returns <code>count</code> , 5

`n` shows how much is left, and `count` shows how many halvings have been done so far, just like a loop's counter. The 5 is passed back unchanged through every waiting call. Note $5 / 2$ is 2, not 2.5.

Reread *4.16.5 Parameters Track the Progress* if your trace differed.

Question 10

B. `total` plays the role of the loop's running total. Each call passes `total + i`, so by the time `i` is 5, the values 3 and 4 have been added: $0 + 3 + 4 = 7$. The parameter values record the progress so far, just as `total` would in the loop's third pass.

- **A** is wrong: this is only the third call (`i` went 3, 4, 5). `total` holds a sum, not a count of calls.
- **C** is wrong: 5 and 6 still have to be added, giving 18.
- **D** is wrong: the next number to add is `i`, which is 5. The parameter `total` holds what has already been added.

Reread 4.16.5 *Parameters Track the Progress* if you chose differently.

Question 11

A. The recursive method adds `n`, `n - 2`, `n - 4`, ... while the value is positive, and returns 0 when it reaches 0 or less. Choice A starts `k` at `n`, subtracts 2 each pass (the changed argument), stops when `k > 0` is false (the base case), and starts `total` at 0 (the base case's value). Running every version for `n = 7` and `n = 8` gives:

Version	<code>n = 7</code>	<code>n = 8</code>
<code>sumEveryOther</code>	16	20
Choice A	16	20
Choice B	16	16
Choice C	28	36
Choice D	8	12

- **B** is wrong: it adds 1, 3, 5, ... no matter what `n` is. For 7 it happens to match (16), but for 8 it gives 16 instead of $8 + 6 + 4 + 2 = 20$. Testing one value is not enough.
- **C** is wrong: it steps by 1, so it adds every number from `n` down to 0, not every other one.
- **D** is wrong: it subtracts 2 **before** adding, so `n` itself is never added and one value at or below 0 is added at the end. For 7 it adds $5 + 3 + 1 + (-1) = 8$.

Reread 4.16.6 *Recursion and Loops Can Replace Each Other* if you chose differently.

Question 12

A model answer: No. Any recursive solution can be rewritten using a loop, and any loop can be rewritten using recursion. For example, `digitSumRecursive` and `digitSumLoop` both return 14 for 4721; the base case `n < 10` corresponds to the loop ending, and the recursive call's argument `n / 10` corresponds to the update `n = n / 10`. Which version to use is a choice about clarity, not about what is possible.

A correct answer must:

- say the classmate is wrong, because every recursive solution has an iterative version and vice versa;
- support it with a matching pair, such as the digit sums, the powers, the countdowns, or the range sums.

Reread 4.16.6 *Recursion and Loops Can Replace Each Other* if your answer differed.

Question 13

A. The calls are `fold(7)`, `fold(5)`, `fold(3)`, and `fold(1)`: four calls. `fold(1)` is a base case (`1 <= 2` is true) and returns 1. Then each waiting call adds 1: `fold(3)` returns 2, `fold(5)` returns 3, and `fold(7)` returns 4.

- **B** is wrong: it forgets to count either the first call from `main` or the base-case call. Both are calls to `fold`.
- **C** is wrong: it assumes the original argument comes back. The base case returns its own `n`, which is 1, not 7.
- **D** is wrong: it misses a call, which also loses one of the `+ 1`s.

Reread 4.16.2 *The Base Case Stops the Recursion* and 4.16.4 *Returning Values Through the Calls* if you chose differently.

Question 14

```
<5
<3
<1
|
1>
3>
5>
```

```
pattern(5)  prints <5, then calls pattern(3) and waits
pattern(3)  prints <3, then calls pattern(1) and waits
pattern(1)  prints <1, then calls pattern(-1) and waits
pattern(-1) -1 <= 0: base case, prints |
pattern(1) resumes: prints 1>
pattern(3) resumes: prints 3>
pattern(5) resumes: prints 5>
```

Line 11 runs on the way down, so the `<` lines appear in call order. Line 13 runs on the way back up, so the `>` lines appear in reverse order, each using its own call's `n`. The base case is `n <= 0`, so stepping from 1 to `-1` is caught safely.

Reread 4.16.3 *Each Call Has Its Own Variables* if your output differed.

Question 15

```
1011
```

Call	<code>n / 2</code>	<code>n % 2</code>	Returns
<code>convert(11)</code>	5	1	<code>101 * 10 + 1 = 1011</code>
<code>convert(5)</code>	2	1	<code>10 * 10 + 1 = 101</code>
<code>convert(2)</code>	1	0	<code>1 * 10 + 0 = 10</code>
<code>convert(1)</code>	base		1

Fill in the "Returns" column from the bottom up: each call needs its recursive call's value before it can multiply by 10 and add its remainder. (The digits 1011 happen to be 11 written in base 2, but you can trace the method without knowing that.)

Reread 4.16.4 *Returning Values Through the Calls* if your answer differed.

Question 16

B. Each call subtracts 4, so the argument reaches exactly 0 only if it starts at 0 or above and is a multiple of 4. The program shows this: `drain(8)` goes 8, 4, 0 and prints `Empty`; `drain(0)` prints `Empty` at once; then `drain(6)` goes 6, 2, -2, ... , skips past 0, and crashes with `StackOverflowError`.

- **A** is wrong: 6 is 0 or more, but 6, 2, -2 jumps over 0.
- **C** is wrong: a negative start such as -4 goes -8, -12, ... , moving away from 0, so it also crashes.
- **D** is wrong: it leaves out 0, which works, and includes values like 6, which do not.

Reread 4.16.2 *The Base Case Stops the Recursion* if you chose differently.

Question 17

Line 7 is the error. The base case returns 0, and that 0 is multiplied into every result on the way back: `factorial(2)` returns $2 \times 0 = 0$, `factorial(3)` returns $3 \times 0 = 0$, and `factorial(4)` returns $4 \times 0 = 0$. The recursion itself is fine; it reaches the base case and stops. The value the base case hands back is wrong.

Corrected line 7:

```
1         return 1;
```

Now the returns are 1, then $2 \times 1 = 2$, then $3 \times 2 = 6$, then $4 \times 6 = 24$, and the program prints 24. The base case value must be one that leaves the product unchanged, which is 1 (for a sum, like `sumEveryOther`, it is 0).

Reread 4.16.4 *Returning Values Through the Calls* if you missed it.

Question 18

405

Call	n % 10	Next built	Next call
<code>reverseDigits(504, 0)</code>	4	$0 * 10 + 4 = 4$	<code>reverseDigits(50, 4)</code>
<code>reverseDigits(50, 4)</code>	0	$4 * 10 + 0 = 40$	<code>reverseDigits(5, 40)</code>
<code>reverseDigits(5, 40)</code>	5	$40 * 10 + 5 = 405$	<code>reverseDigits(0, 405)</code>
<code>reverseDigits(0, 405)</code>	base		returns 405

The two parameters track progress: `n` holds the digits still to move, and `built` holds the reversed digits so far, like the variables of a loop. The middle 0 is kept because `built` is multiplied by 10 before 0 is added. The base case returns `built`, and every waiting call passes 405 straight back.

Reread 4.16.5 *Parameters Track the Progress* if your trace differed.

Question 19

D. Work from the base cases up, the same way the trace in Putting it together ends: `count(0)` and `count(1)` return 1. Then

- `count(2) = count(1) + count(0) + 1 = 1 + 1 + 1 = 3`
- `count(3) = count(2) + count(1) + 1 = 3 + 1 + 1 = 5`

- $\text{count}(4) = \text{count}(3) + \text{count}(2) + 1 = 5 + 3 + 1 = 9$

So $\text{count}(4)$ returns 9.

- **A** is wrong: 5 is $\text{count}(3)$, one level short, or the result of $\text{ways}(4)$ from Putting it together, which has no $+ 1$.
- **B** is wrong: it adds 1 only once at the top instead of in every recursive case (for example $3 + 3 + 1$).
- **C** is wrong: it leaves out the final $+ 1$ in $\text{count}(4)$.

Reread *Putting it together* and 4.16.4 *Returning Values Through the Calls* if you chose differently.