

Chapter 7. Repetition

In Chapter 1 we saw that algorithms deploy sequence, selection and repetition statements in combination to specify computations, in Chapters 2-4 we used sequential execution and then in Chapter 5 introduced selective execution. In this chapter we introduce the final control mechanism, *repetitive* execution, in which a set of instructions is executed repeatedly until some condition terminates the repetition.

This chapter introduces the use of basic repetition statements in Processing. It also introduces recursion, an alternate repetitive control structure.

7.1. Example: Drawing Curves

There are only three basic control structures: sequence, selection and repetition. The power of algorithms comes from the ability to use of these control structures in creative combinations to specify computations that achieve a wide variety of potential goals. This chapter will focus on a rather simple goal, but the same principle serves as the foundation for all modern information and communication technology.

Consider the goal of plotting curves as specified by mathematical functions. For example, Figure 7-1 shows two curves: the square-root of x and x -squared, and one straight line, all plotted from 0.0 to 1.0 on both the x and the y axes. To be useful, we'd like the application to be general enough to plot any one of a variety of functions, using arbitrary boundaries on the axes.

Our vision is to build a function plotter that plots lines according to a function or functions chosen by the user from a predefined list. This will require that we implement the following features:

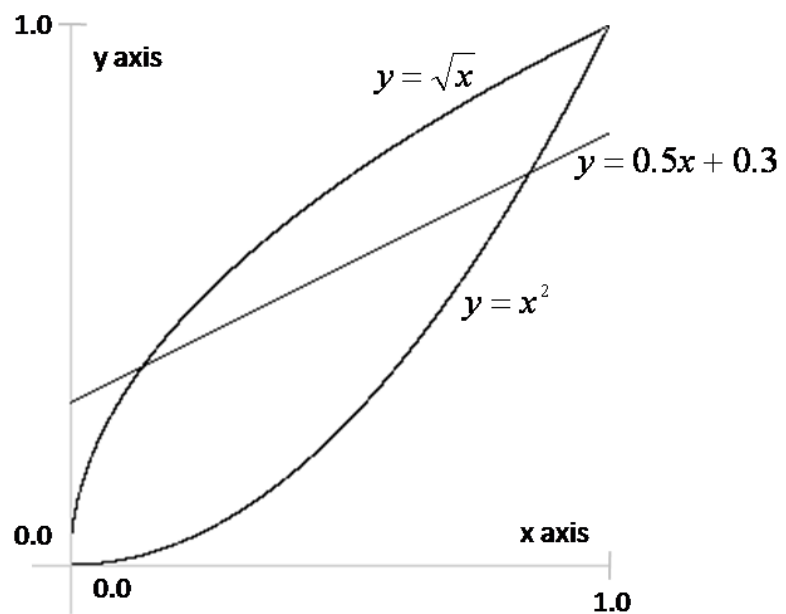


Figure 7-1. A plot of three functions of x

- The user should be able to choose a function to plot from a predefined list. Implementing this feature requires *selective* execution based upon a user choice, as discussed in Chapter 5.
- The application should plot all the points for the function chosen by the user as bounded by upper and lower limits on the x and y coordinates. Chapter 2 discussed the `line()` method, which

could be used to draw the straight line in the example (i.e., $y = 0.5x \times 0.3$) and the x and y axes, but it would not work well for the curves. Chapter 2 also showed how to draw lines and curves one pixel at a time using the `point()` method, but this approach would be too tedious to program using sequence alone. Implementing curve plotting, therefore, requires *repetitive* execution.

This chapter shows how the control structures for sequence, selection and repetition can be used in combination to specify a wide variety of computations.

7.2. Repetition

This section introduces the use of repetitive execution in Java. Repetition allows programs to state, in a concise way, a repetitive or iterative behavior such as counting or drawing multiple points along a curve. Java provides the **while**, **do** and **for** statements to implement looping; this section discusses each in turn.

7.2.1. The while Loop

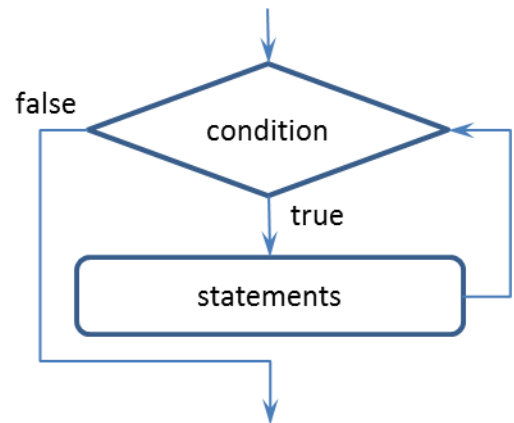
Consider the task of printing the integers from some value, say 5, down to zero. One approach to implementing this behavior would be to write a sequence of print statements such as:

Code	Output
<code>System.out.println(5);</code>	5
<code>System.out.println(4);</code>	4
<code>System.out.println(3);</code>	3
<code>System.out.println(2);</code>	2
<code>System.out.println(1);</code>	1
<code>System.out.println("Lift off!");</code>	Lift off!

While this code arguably solves the problem as stated, it is clearly not a scalable solution. For example, counting down from 1000 would take 1001 statements – unacceptable! A better approach to implementing this behavior is to use a **while** loop, which has the following basic form:

```
while (condition) {
    statements
}
```

Here, the statements in the block are executed repeatedly so long as the condition is true. This pattern can be visualized as shown on the right.



We can use this statement to implement a countdown from 5 to 0 as follows:

Code	Output
<pre>int count = 5; while (count > 0) { System.out.println(count); count--; } System.out.println("Lift off!");</pre>	5 4 3 2 1 Lift off!

This code starts by declaring an integer variable **count** and initializing it to 5. It then executes a **while** loop that repeatedly prints and then decrements the value of **count**. The loop continues executing until the loop continuation condition, **count > 0**, is no longer true, which happens after five iterations because the loop statements decrement **count** by one each time. When the loop is complete, it prints the message “Lift off!”. One advantage of this looping statement is that it can easily be modified to count down from 1000. Further, and perhaps more importantly, the code is simpler and clearly indicates the pattern that it is implementing - print the numbers down from **count** to 0.

The general form of the while loop is as follows:

while Statement Pattern

```
while (condition)
    statement
```

- condition is a Boolean expression whose value dictates when the looping terminates – so long as condition is true, the loop continues; when condition becomes false, the loop stops;
- statement is a simple statement or block of statements.

We must be careful to ensure that the loop termination condition eventually becomes false. For example, if we copied the previous code but forgot to include the decrement statement, as shown here, we'd clearly have a problem:

Code	Output
<pre>int count = 5; while (count > 0) { System.out.println("I'm stuck in this loop."); }</pre>	I'm stuck in this loop. I'm stuck in this loop. I'm stuck in this loop. I'm stuck in this loop. <i>...and so forth, forever...</i>

This loop would not terminate appropriately. Because **count** starts at 5 and never changes, the condition **count > 0** would remain true forever. This problematic form of a loop is sometimes called an *infinite loop*.

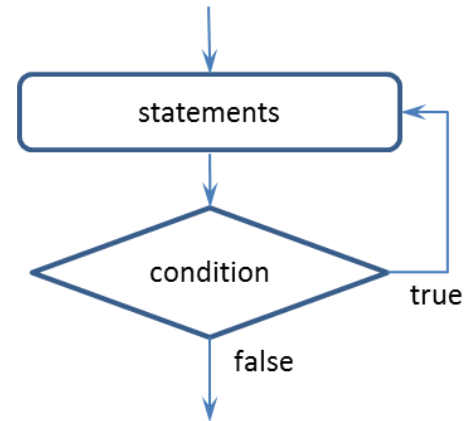
7.2.2. Using the do Statement

As discussed in the previous section, the **while** loop's condition is checked before executing the loop statements. There are cases in which it is more convenient to check the condition after executing the loop statements. The **do** loop is designed for these situations.

The **do** loop has the following basic form:

```
do {  
    statements  
} while (condition);
```

Here, the statements in the block are executed once and then the condition is checked. Looping continues so long as the condition is true. Note that the semi-colon after the while condition is required in this pattern. This pattern can be visualized as shown on the right.



Consider the problem of printing out the sequence of squares, 1^2 , 2^2 , 3^2 , etc., going as high as the user chooses to go. We'd like to print out 1^2 for sure, then ask the user if they'd like to go on to the next and continuing in that manner until the user chooses to quit. We can use the **do** loop to implement this behavior as follows:

Code

```
Scanner keyboard = new Scanner(System.in);  
  
System.out.println("Welcome to the Squares Program.");  
int x = 1;  
String response;  
do {  
    // Print x-squared.  
    System.out.print(x + "^2 is " + (x * x));  
    x++;  
  
    // Determine whether or not to go on.  
    System.out.print("\tcontinue? (y or n): ");  
    response = keyboard.nextLine();  
} while (response.equals("y") || response.equals("Y"));  
  
System.out.println("finis");
```

Output

```
Welcome to the Squares Program.  
1^2 is 1    continue? (y or n): y  
2^2 is 4    continue? (y or n): y  
3^2 is 9    continue? (y or n): y  
4^2 is 16   continue? (y or n): y  
5^2 is 25   continue? (y or n): y  
6^2 is 36   continue? (y or n): n  
finis
```

In this program, the **do** loop serves our purposes better than the **while** loop would. The do executes its statements once at the beginning and then starts checking its condition, which worked well in this case.

The general form of the **do** loop is as follows:

do Statement Pattern

```
do
  statement
while (condition)
```

- condition is a Boolean expression whose value dictates when the looping terminates – so long as condition is true, the loop continues; when condition becomes false, the loop stops;
- statement is a simple statement or block of statements.

7.2.3. Forever Loops

In the previous two sections, the while and the do loop conditions are checked before and after executing the loop statements respectively. There are some cases in which we'd like to check the condition at arbitrary times during the execution of the loop statements, neither at the beginning nor the end. For example, consider the problem of adding up an arbitrary list of integers input by the user. The program doesn't know how many integers there will be, so we'd like to implement an algorithm like this:

Algorithm:

1. Print a welcome message.
2. Initialize a running sum to 0.
3. Loop
 - a. Print a message asking for an integer.
 - b. Read an integer from the keyboard.
 - c. If the integer just read is the sentinel value
 - i. Stop looping.
 - d. Add the new integer to the running sum.
4. Print the running sum.

In this algorithm, the determination of when to stop the loop is made in the middle of the loop statements rather than at the very beginning. That is, the program needs to read an integer from the keyboard (i.e., steps 3a and 3b) before determining whether the loop should stop. Then, if it is not time to stop, the program must perform some additional processing before continuing with the loop (i.e., step 3d).

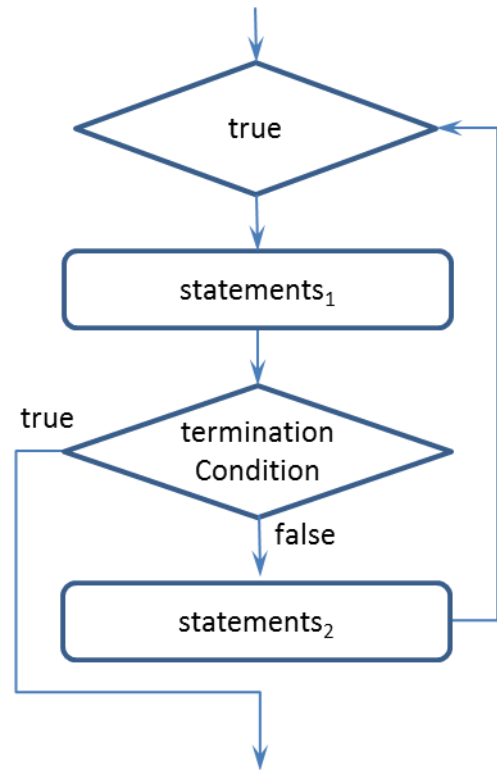
Checking the termination condition in the middle of the loop requires a more general form of looping called the *forever loop*, which has the following basic form:

```

while (true) {
    statements1
    if (terminationCondition) {
        break;
    }
    statements2
}

```

Here, the while condition is set to **true**, which, of course, is always true. Thus, the **while** loop will continue forever. The loop is terminated using a **break** statement, which terminates the current control statement, in this case the **while** loop, and transfers control to immediately enclosing statements, in this case the block containing the **while** statement. This pattern can be visualized as shown on the right.



Note that this algorithm requires the use of a *sentinel* value to determine when to stop processing its input. A sentinel is a “special” value that signals the end of the input values. It must be a value that is of the same type as the legal input values but that is clearly distinguishable from those legal values. For a summation of positive integers, for example, the sentinel value can be -1. Then, if the user enters the sequence 1, 2, 3, -1, the program should sum up 1+2+3 stopping when it sees the -1.

We can use this forever loop mechanism with an explicit termination condition to implement the summation algorithm specified above as follows:

Code

```

final int SENTINEL = -1;
Scanner keyboard = new Scanner(System.in);

System.out.println("Welcome to the Summation Program, please enter some integers.");
int runningSum = 0, x;
while (true) {
    // Read the next integer value (or the sentinel value).
    System.out.print("integer (or " + SENTINEL + " to quit): ");
    x = keyboard.nextInt();

    // When the sentinel is read, terminate the loop.
    if (x == SENTINEL) {
        break;
    }

    // Update the running sum.
    runningSum += x;
}
System.out.println("Summation: " + runningSum);

```

Output

```
Welcome to the Summation Program, please enter some integers.
integer (or -1 to quit): 2
integer (or -1 to quit): 5
integer (or -1 to quit): 1
integer (or -1 to quit): -1
Summation: 8
```

This code defines a constant for the sentinel value, **SENTINEL**, which is used throughout the program. It uses a forever loop that terminates execution using **break** when the sentinel value is entered.

This code uses a standard approach to *accumulating* values in which an accumulator variable, in this case **runningSum**, is initialized to 0 and is then updated repeatedly until all the input has been processed. The resulting value of the accumulator is the program's output value. It's important to initialize an accumulator variable to a suitable value, in this case 0.

7.2.4. Using the **for** Statement

It is frequently the case that programs must perform counting tasks. For example, we might want a program to count from 1 to 10. We can do this using a **while** loop as follows:

Code	Output
<pre>int i = 1; while (i <= 5) { System.out.println(i); i++; }</pre>	<pre>1 2 3 4 5</pre>

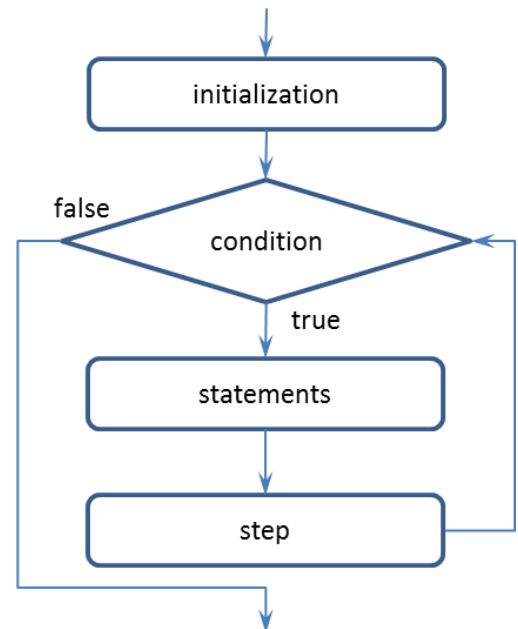
This code initializes an index variable, **i**, to the starting value, 1, compares its value to the ending value, 5, and repeatedly prints and increments its value. This indexing pattern is so common, that the **for** loop was developed to support it explicitly. The pattern for the **for** loop is as follows:

```
for (initialization; condition; step) {
    statements
}
```

The **for** statement has special sections that specify the initialization statement, the loop condition and the increment or step expression. The flow of execution can be visualized as shown to the right.

The **while** statement in the code shown above can be re-expressed as an equivalent but more concise **for** loop as shown here:

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```



This program has the same output, but streamlines the specification of the initialization, condition and step. The relationship between the two formulations is shown here:

```

int i = 1
while ( i <= 5 ) {
    System.out.println(i);
    i++;
}

for ( int i = 1 ; i <= 5 ; i++ ) {
    System.out.println(i);
}

```

The general form of the **for** loop is as follows:

for Statement Pattern

```
for(initializationExpression; condition; changeExpression)
    statement
```

- ***initializationExpression*** can be any Processing expression, but is generally used to declare and initialize a loop counting variable (frequently named *i*);
- ***condition*** is a Boolean expression that dictates how long the loop keeps going - It is commonly used to compare the loop counter with some limiting value;
- ***changeExpression*** can be any Processing expression, but is generally used to increment (or decrement) the loop counter;
- ***statement*** is a simple statement or a block of statements.

7.2.5. Choosing the Right Loop

As shown in the previous sections, there are a number of choices for implementing looping behavior, and it's entirely possible to re-implement a loop that uses one of the forms into any of the other forms. It is, therefore, important to be able to choose the appropriate form for each situation.

1. To count or not to count – For situations in which counting or indexing are important, the **for** loop is usually the best choice.
2. If counting is not a significant part of the problem, then one of the other more general loops is likely to be preferable:
 - a. Pre-testing – Problems in which it is most natural to test the loop condition at the beginning of the loop should use the **while** loop.
 - b. Post-testing – Problems in which it is most natural to test the loop condition at the end of the loop should use the **do** loop.
 - c. Otherwise, the forever loop with manual loop termination, using **break** (or perhaps **return** or even **exit()**), is likely the best choice.

As an example, consider the problem of computing the factorial n , i.e., $n!$. This problem requires that we multiply the values of 1, 2, ..., n and thus we require an index whose value counts from 1 to n . The following algorithm can be used to solve this problem:

Given

- An integer value n .

Algorithm:

1. Set runningProduct = 1.
2. Loop for i set to values from 1 up to n
 - a. Set runningProduct = runningProduct * i .
3. Print the running sum.

The central feature of this algorithm is the index value that counts from 1 to n . This leads us to choose the **for** loop. The algorithm also uses an accumulator, as was done with the summation program discussed above, but this time the initial value is 1, the identity value for multiplication. The following program implements this algorithm:

Code	Output
<pre>final int N = 3; int runningProduct = 1; for (int i = 1; i <= N; i++) { runningProduct *= i; } System.out.println(runningProduct);</pre>	<p>6</p>

7.2.6. Nested Loops

The power of algorithms is based on the ability to use the basic control structures in combination. One particularly useful combination of control structures is when a loop is used inside of another loop. This is called *nested loops*. Situations in which there are two dimensions of looping that must be integrated are natural situations in which to apply nested loops.

Consider the problem of printing a multiplication table. Here, the program must multiply all possible combinations of two indexes and print the result in a two-dimensional table. This is a natural situation in which to apply nested **for** loops. The following algorithm specifies a solution to this problem.

Given

- An integer value n (the width/height of the desired table)

Algorithm:

4. Set runningProduct = 1.
5. Repeat for a counter i ranging from 1 to n :
 - a. Set runningProduct = runningProduct * i .
6. Print the running sum.

The central feature of this algorithm is the index value that counts from 1 to n . This leads us to choose the **for** loop. The algorithm also uses an accumulator, as was done with the summation program discussed

above, but this time the initial value is 1, the identity value for multiplication. The following program implements this algorithm for a 5X5 table:

Code	Output
<pre>final int N = 5; for (int i = 1; i <= N; i++) { for (int j = 1; j <= N; j++) { System.out.print("\t" + (i * j)); } System.out.println(""); }</pre>	<pre>1 2 3 4 5 2 4 6 8 10 3 6 9 12 15 4 8 12 16 20 5 10 15 20 25</pre>

Note how this code puts one `for` loop inside of the other, being careful to use two different index variables, `i` and `j`. For each execution of the outer loop, that is for each value of `i`, the inner loop runs completely, that is for all values of `j`. The inner loop prints one row of values; the outer loop prints each of the rows.

7.3. Repetition in Processing

Because Processing is based on Java, the loop variants discussed so far in this chapter work equally well in Processing. Unfortunately, Processing does not support user input, so the examples discussed above that use the Java Scanner to read user input cannot be run in Processing. In addition, notice that Processing supports two forms of repetitive behavior:

- The explicit looping constructs discussed in this chapter through which programmers implement their own repetitive behaviors;
- Processing's animation features discussed in previous chapters in which Processing drives the repetitive display of animation frames and repetitive handling of user input.

It's important not to confuse these two types of repetition. Each has its use, and they can be deployed together, separately or not at all.

7.3.1. Using the `for` Loop in Processing

Consider the following algorithm, which draws a series of points on the output canvas.

Given:

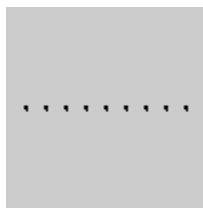
- `count` is set to an integer representing the desired number of points for the dotted line.

Algorithm:

1. Repeat for a counter `i` ranging from 1 to `count`:
 - a. Draw a point at coordinates $(i * 10, 50)$.

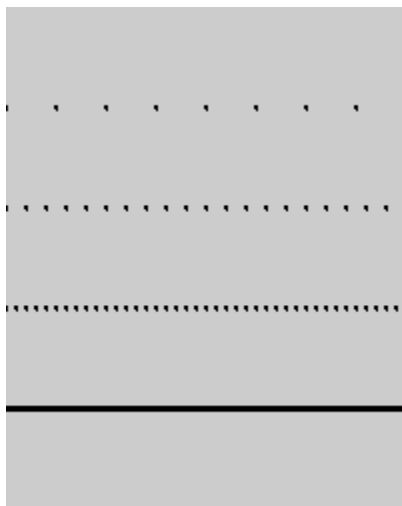
This algorithm crystallizes three key elements of horizontal dotted lines: (1) Dotted lines are made up of single repeated element, the point, so the algorithm includes a single operation that draws a point and places that operation in a repetition statement; (2) the value of the `y` coordinate of the point operation is

hard-coded to 50; and (3) the x coordinate is set to values starting a 10 and increasing by 10 on each repetition. The following code implements this algorithm.



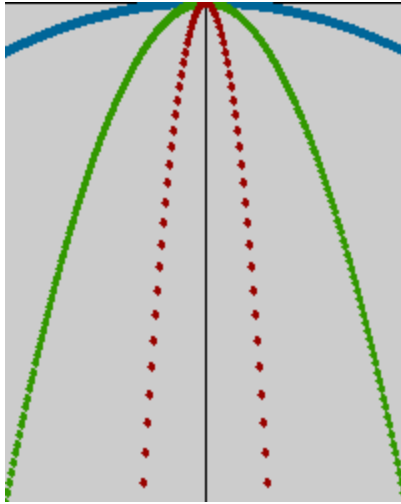
```
strokeWeight(3);  
  
// Draw a dotted line using a counting for loop.  
for (int i = 1; i <= count; i++) {  
  point(i * 10, 50);  
}
```

The `for` loop is powerfully concise statement of repetitive, counting behavior. The following program combines the `for` loop with methods to create a tool for drawing various sorts of dotted lines.



```
final int WIDTH=200, HEIGHT=250;  
  
void setup() {  
  size(WIDTH, HEIGHT);  
  //background(255);  
  strokeWeight(3);  
  
  drawDottedLine(25, 50);  
  drawDottedLine(10, 100);  
  drawDottedLine(5, 150);  
  drawDottedLine(1, 200);  
}  
  
void drawDottedLine(int increment, int y) {  
  for (int i = 0; i < WIDTH; i+=increment) {  
    point(i, y);  
  }  
}
```

The `drawDottedLine()` method is a useful abstraction for drawing dotted lines because it can be called with arguments that specify different sorts of dotted lines, including the four shown above. The call to `drawDottedLine(1, 200)` plots a straight line segment with no gaps. The following example uses this method as a model to draw parabolae of different shapes and colors.



```

final int WIDTH=200, HEIGHT=250;

void setup() {
  size(WIDTH, HEIGHT);

  drawAxes();
  drawParabola(.01, color(0, 102, 153)); // blue
  drawParabola(0.1, color(50, 152, 0)); // green
  drawParabola(1.0, color(152, 3, 0)); // red
}

void drawAxes() {
  strokeWeight(1);
  stroke(0); // Draw the axes in black.
  line(WIDTH/2, 0, WIDTH/2, HEIGHT);
  line(0, 0, WIDTH, 0);
}

void drawParabola(float c, int myColor) {
  strokeWeight(5);
  stroke(myColor);
  for (float x = -WIDTH/2; x < WIDTH/2; x++) {
    point(x + WIDTH/2, pow(x, 2) / 4.0 * c);
  }
}

```

The `drawParabola()` method draws a parabola specified by the given constant using the given color. Its `for` loop assigns successive values for the x axis, x , and computes the appropriate value for the y axis using the general parabolic formula, $x^2/4c$. In order to show the full parabola, the `for` loop initializes x to $-width/2$, loops through $width/2$, and sets the x value of each point to $x + width/2$; this slides the y axis to the center of the output pane. In order to follow Processing's convention of having the y values grow toward the bottom of the output pane, `drawAxes()` places the x axis at the top of the output pane and `drawParabola()` draws the parabola upside-down.

7.3.2. Using Nested `for` Loops in Processing

Nested `for` loops are useful for working with multi-dimensional data such as the pixel addresses on the Processing output pane. Consider the goal of drawing a point with a random color at each screen pixel of the output frame. An algorithm for doing this is shown here.

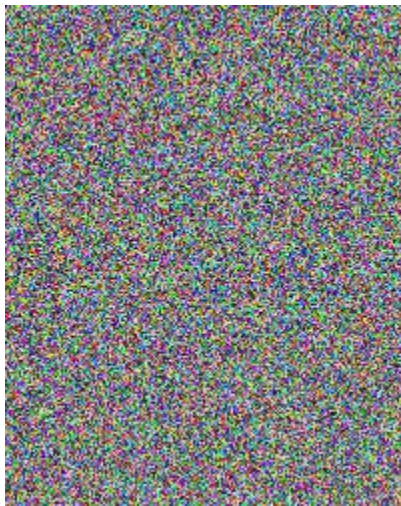
Given:

- We know the width and height of the output frame.

Algorithm:

1. Repeat for a counter i ranging from 0 to $width - 1$:
 - a. Repeat for a counter j ranging from 0 to $height - 1$:
 - i. Draw a point with a random color at coordinates (i, j) .

This algorithm uses a loop inside of a loop to visit each screen pixel exactly once. The following program implements this algorithm.



```
final int WIDTH = 200, HEIGHT = 250;

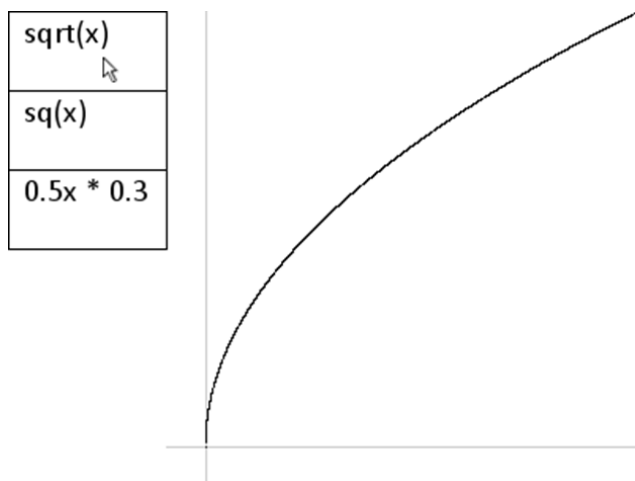
void setup() {
  size(WIDTH, HEIGHT);

  for (int x = 0; x < WIDTH; x++) {
    for (int y = 0; y < HEIGHT; y++) {
      stroke(random(0,255),
             random(0,255),
             random(0,255));
      point(x, y);
    }
  }
}
```

This example uses nested `for` loops to address each pixel on the Processing output pane. The outer loop uses the variable `x` to represent the `x` coordinate ranging from 0 to the width of the output frame, and the inner loop uses the variable `y` to represent the `y` coordinate ranging from 0 to the height of the output frame. The two statements in the inner `for` loop are executed exactly `width × height` times, once for each screen pixel in the output.

7.3.3. Revisiting the example

The chapter example must plot functions. On this iteration, the prototype plots each function that the user chooses. The choice mechanism is based on the graphical choosers implemented in Chapter 5. In the example shown here, the user has chosen to graph $y = \sqrt{x}$.



```

/* functionChooserPlotter plots the selected function from 0 to 1.
 *
 * @author kvlinden
 * @version 14july2009
 */

final int SQRT = 0, SQ = 1, LINE = 2;
int functionMode = SQRT;

int menuWidth = 100, menuHeight = menuWidth/2, xAxisWidth = 300,
    width = menuWidth + xAxisWidth, height = xAxisWidth;
int xAxis = height-25, yAxis = 25;
float xMax = 1.0, scale = (xAxisWidth - yAxis) / xMax,
    xMin = -yAxis / scale;

void setup() {
    size(width, height);
    background(255);
    textFont(loadFont("Calibri-20.vlw"));
}

void draw() {
    background(255);
    drawMenu();
    drawAxes(xAxis, yAxis);
    plotFunction();
}

// This method plots the function for each value on the x axis.
void plotFunction() {
    stroke(0);
    for (float x = 0; x <= xMax; x += 0.0001) {
        point(translateX(x), translateY(computeY(x)));
    }
}

// This method converts the raw x coordinate to be plotted into the
// appropriate canvas value.
float translateX(float value) {
    return value * scale + menuWidth + yAxis;
}

// This method inverts the y value for plotting on the canvas.
float translateY(float value) {
    return value * -1 * scale + xAxis;
}

// This method computes the appropriate function value for x depending
// upon which function the user has chosen.
float computeY(float x) {

```

```

switch(functionMode) {
case SQR:
    return sqrt(x);
case SQ:
    return sq(x);
case LINE:
    return 0.5 * x + 0.3;
default:
    return 0.0;
}
}

// This method draws the function chooser menu on the left of the
// canvas.
void drawMenu() {
    fill(255);
    rect(0, 0, menuWidth, menuHeight);
    rect(0, menuHeight, menuWidth, menuHeight);
    rect(0, menuHeight * 2, menuWidth, menuHeight);
    fill(0);
    text("sqrt(x)", 10, 20);
    text("sq(x)", 10, menuHeight + 20);
    text("0.5x * 0.3", 10, menuHeight*2 + 20);
}

// This method draws the axes in a location that leaves room for the
// chooser buttons.
void drawAxes(int x, int y) {
    stroke(200);
    line(menuWidth, xAxis, width, xAxis); // x axis
    line(menuWidth + yAxis, 0, menuWidth + yAxis, height); // y axis
}

// This method determines which function the user has selected.
void mouseClicked() {
    if (mouseX < menuWidth) {
        if ((mouseY >= 0) && (mouseY < menuHeight)) {
            functionMode = SQR;
        }
        else if ((mouseY >= menuHeight) && (mouseY < menuHeight * 2)) {
            functionMode = SQ;
        }
        else if ((mouseY >= menuHeight * 2) && (mouseY < menuHeight * 3))
        {
            functionMode = LINE;
        }
    }
}
}

```

This program is largely copied from the previous iteration, but implements the `plotFunction()` using the curve plotting approach discussed in this section. One key change is that it plots the function right-side up, with the x axis going along the bottom of the output window. It does this by implementing

two translate methods, `translateX()` and `translateY()`, which receive the `x` and `y` values from their calling programs and convert them into pixel coordinates appropriate for the actual orientation of the `x` and `y` axes. `translateX()` moves `x` values over to accommodate the menu on the left side, and `translateY()` converts small `y` values for the function into large pixel coordinate values near the bottom of the screen. These routines also scale the `x` and `y` values to match the graph limits specified by `xMax` and `xMin`.

7.4. Using the Control Structures in Combination

As discussed in the introduction to this chapter, the real power of the three basic control structures comes when they are nested in innovative ways. We've already seen `if` statements nested within `if` statements and `for` statements nested within `for` statements, but we can nest any control statement within any other control statement.

Consider the goal of plotting the sine function on a labeled `x` axis. In the chapter example, we used a `for` loop to implement the `plotFunction()` method, which plotted the values of a function between two specified boundaries. Unfortunately, that method does not label the `x` axis. This task requires selective behavior that is best implemented within the `for` loop itself. This behavior can be specified using the following algorithm.

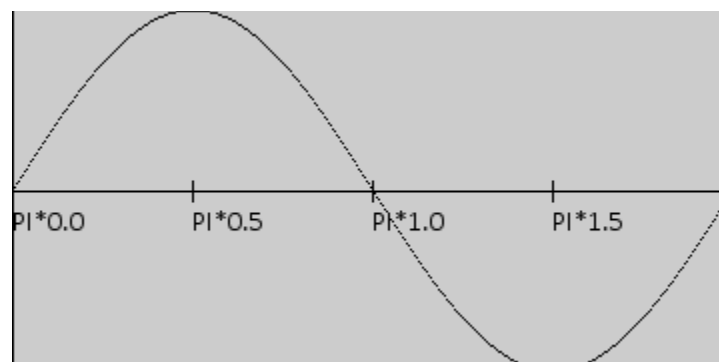
Given:

- `WIDTH` and `HEIGHT` are set to the width and height of the output frame respectively.

Algorithm:

1. Draw the `x` axis at `y = height/2`.
2. Draw the `y` axis at `x = 0`.
3. Repeat for a counter `i` ranging from 0 to `width - 1`:
 - a. If the `x` value is a multiple of 90 degrees:
 - i. Draw a hash mark on the `x` axis.
 - ii. Label the hash mark appropriately.
 - b. Plot a point at coordinates $(x, \sin(x))$, being sure to translate the `Y` value appropriately.

This algorithm solves this particular problem by nesting selection and sequence within a repetition loop. It can be implemented as follows.




```

/* SineWave plots the sine wave from 0 to 2*PI.
 *
 * @author kvlinden
 * @version 14july2009
 */
int width = 360, height = width/2;

void setup() {
  size(width, height);
  textFont(loadFont("Calibri-14.vlw"));
  noLoop();
  fill(0);
}

void draw() {
  line(0, height/2, width, height/2); // x axis
  line(0, 0, 0, height); // y axis
  for (int x = 0; x <= width; x++) {
    if (x % 90 == 0) {
      line(x, height/2-5, x, height/2+5);
      text("PI*" + radians(x)/PI, x, height/2 + 20);
    }
    point(x, translateY(sin(radians(x)) * height/2));
  }
}


// This method inverts the y value for plotting on the canvas.
float translateY(float value) {
  return value * -1 + height/2;
}

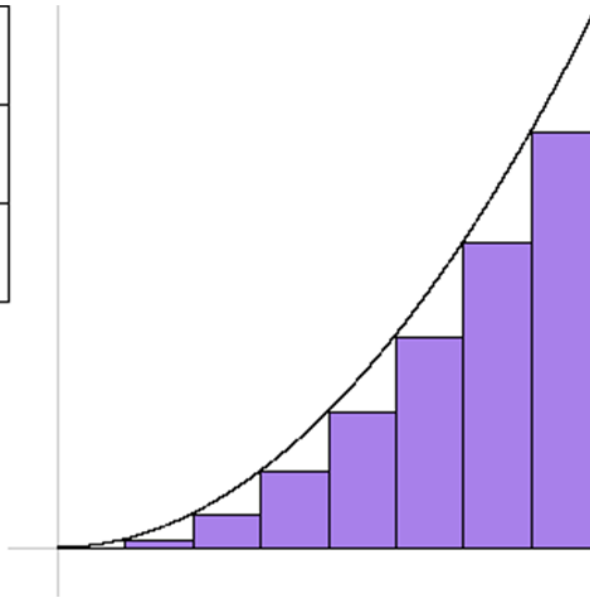
```

This program uses a combination of sequence, selection and repetition to plot a sine wave and annotate the x axis. In particular, the `draw()` method uses a sequence of statements including an `if` statement in a `for` statement.

7.5. Revisiting the Example

As a final iteration in the chapter example, we now use the function plotter to animate the use of a series of progressively smaller rectangles in the definite integral. The code computes the sum of areas of all the rectangles. The completed program along with one snapshot of its output is shown here.

\sqrt{x}
$\text{sq}(x)$ 
$0.5x * 0.3$



```

/**
 * AreaApproximator animates the approximation of the definite integral of
 * f(x)=sqrt(x) from 0 to 1.
 *
 * @author jnyhoff, nyhl, kvlinden, snelesen
 * @version Fall, 2011
 */

final int MENU_WIDTH = 100, MENU_HEIGHT = MENU_WIDTH/2, X_AXIS_WIDTH = 300;
final int WIDTH = MENU_WIDTH + X_AXIS_WIDTH, HEIGHT = X_AXIS_WIDTH;
final int X_AXIS = HEIGHT-25, Y_AXIS = 25;
final float X_MAX = 1.0,
          SCALE = (X_AXIS_WIDTH - Y_AXIS) / X_MAX, X_MIN = -Y_AXIS / SCALE;
final int SQRT = 0, SQ = 1, LINE = 2;

int functionMode;
boolean animating;
int numRects;
float area;

void setup() {
  size(WIDTH, HEIGHT);
  textFont(createFont("Calibri",20));
  numRects = 1;
  animating = false;
  functionMode = SQ;
  frameRate(4);
}

void draw() {
  background(255);
  drawMenu();
  drawAxes(X_AXIS, Y_AXIS);
  plotFunction();
  if (animating) {
    area = drawApproximationRectangles(numRects++);
  }
}

```

```

println(area);
if (numRects > 100) noLoop();
}

float drawApproximationRectangles(int n) {
  float result = 0;
  float x, y, rectangleWidth = X_MAX / n;
  for (int i = 0; i < n; i++) {
    x = i * rectangleWidth;
    y = computeY(x);
    plotRectangle(x, y, rectangleWidth, y);
    result += rectangleWidth * y;
  }
  return result;
}

void plotRectangle(float x, float y, float width, float height) {
  fill(168, 128, 234);
  rect(translateX(x), translateY(y),
    translateDistance(width), translateDistance(height));
}

void plotFunction() {
  stroke(0);
  for (float x = 0; x <= X_MAX; x += 0.0001) {
    point(translateX(x), translateY(computeY(x)));
  }
}

float translateX(float value) {
  return value * SCALE + MENU_WIDTH + Y_AXIS;
}

float translateY(float value) {
  return value * -1 * SCALE + X_AXIS;
}

float translateDistance(float value) {
  return value * SCALE;
}

float computeY(float x) {
  switch(functionMode) {
  case SQRT:
    return sqrt(x);
  case SQ:
    return sq(x);
  case LINE:
    return 0.5 * x + 0.3;
  default:
    return 0.0;
  }
}

void drawMenu() {
  fill(255);
  rect(0, 0, MENU_WIDTH, MENU_HEIGHT);
}

```

```

    rect(0, MENU_HEIGHT, MENU_WIDTH, MENU_HEIGHT);
    rect(0, MENU_HEIGHT * 2, MENU_WIDTH, MENU_HEIGHT);
    fill(0);
    text("sqrt(x)", 10, 20);
    text("sq(x)", 10, MENU_HEIGHT + 20);
    text("0.5x * 0.3", 10, MENU_HEIGHT*2 + 20);
}

void drawAxes(int x, int y) {
    stroke(200);
    line(MENU_WIDTH, X_AXIS, WIDTH, X_AXIS); // x axis
    line(MENU_WIDTH + Y_AXIS, 0, MENU_WIDTH + Y_AXIS, HEIGHT); // y axis
}

void mouseClicked() {
    if (mouseX < MENU_WIDTH) {
        if ((mouseY >= 0) && (mouseY < MENU_HEIGHT)) {
            functionMode = SQRT;
        }
        else if ((mouseY >= MENU_HEIGHT) && (mouseY < MENU_HEIGHT * 2)) {
            functionMode = SQ;
        }
        else if ((mouseY >= MENU_HEIGHT * 2) && (mouseY < MENU_HEIGHT * 3)) {
            functionMode = LINE;
        }
        numRects = 1;
    }
}

void mousePressed() {
    animating = true;
    numRects = 1;
}

void mouseReleased() {
    animating = false;
}

```

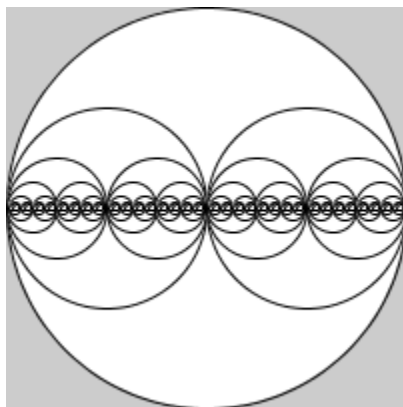
This program animates the construction of the area approximating rectangles, starting with one rectangle, and proceeding until the user releases the mouse. The user can change functions and then start the animation again with the new curve.

7.6. Recursion¹

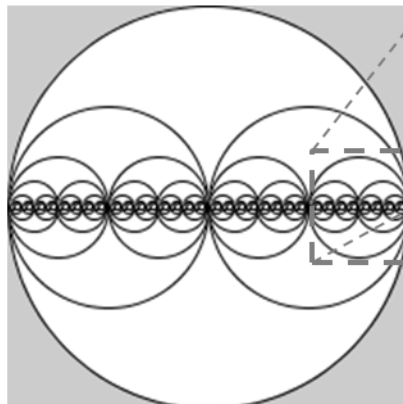
A Google search on the word “recursion” produces the usual list of websites you’d expect, but it also generates the irritatingly circular suggestion “**Did you mean:** [recursion](#)”.² Of course you meant recursion, that’s why you entered that search term in the first place. This is Google’s rather geeky way of demonstrating the meaning of the term recursion, which refers to the method of defining an entity in a self-referential manner. An older, textual version of Google’s joke is shown in the following fanciful dictionary entry:

Recursion (n)
See recursion.

The notion of recursion has a deep and fruitful history in a wide range of fields, including mathematics and linguistics. In computer science, *recursion* refers to the ability of a method to call itself. Such a method is called a *recursive method*.



Images can exhibit this same self-referential or self-similar structure as well. Such recursive images are called *fractal images*. Consider the collection of circles shown to the left. At first glance, this looks like a geometric sort of multi-eyed insect, which you might draw using a repetitive algorithm that repeats once for each eye, getting progressively smaller as the repetition goes on. Writing such an algorithm would definitely be possible, but it would be rather difficult to get the pattern for the sizes and locations of all the circles just right.

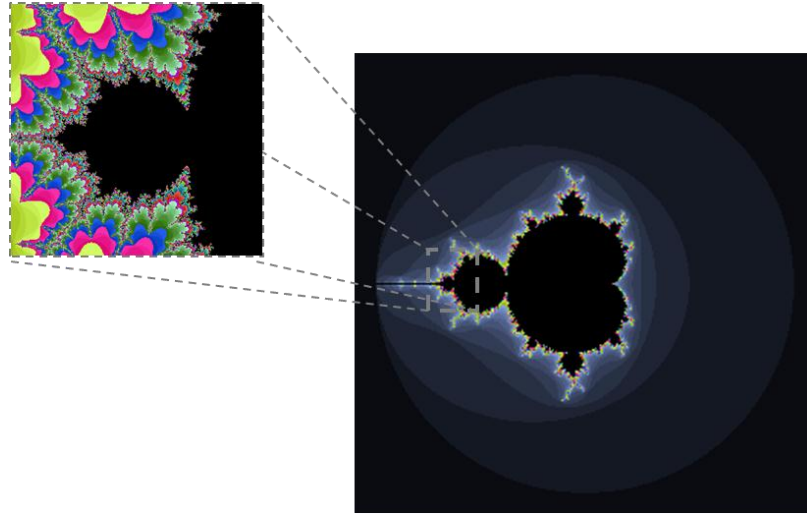


On closer inspection, however, you can see a self-referential pattern in which every circle in the image has two smaller circles inside of it. This pattern repeats itself as the circles get smaller and smaller. It’s a rather simple pattern really. Draw every circle the same way, with a large outer circle and then two smaller inner circles. Theoretically, this recursive pattern could go on getting smaller and smaller forever; it’s only the limited by the resolution of the display window that forces us to stop after seven or eight levels of recursion.

¹ Portions of this section are based on material from Adams, Nyhoff and Nyhoff, *Java: An Introduction to Computing*, Prentice Hall, 2001.

² See <http://googlesystem.blogspot.com/2009/07/google-helps-you-understand-recursion.html> for Google’s explanation.

The Mandelbrot set is another more famous fractal that exhibits a self-referential pattern. As shown here, the iconic snowman images repeat themselves over and over throughout the image.



7.6.1. Computing the Factorial Function

To illustrate the basic idea of recursion, we consider a simple, non-graphical example – the problem of calculating factorials. The factorial of n , denoted as $n!$, is a well-known mathematical function that is defined as follows:

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ 1 \times 2 \times \dots \times n & \text{if } n > 0 \end{cases}$$

For example:

$$0! = 1$$

$$1! = 1$$

$$2! = 1 \times 2 = 2$$

$$3! = 1 \times 2 \times 3 = 6$$

$$4! = 1 \times 2 \times 3 \times 4 = 24$$

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

We can design a method that uses a repetitive algorithm to compute the factorial using the techniques discussed earlier in this chapter.

Given:

- n is a non-negative integer.

Algorithm (for `computeFactorial()`):

- Receive an integer n from the calling program.
- Set `result = 1`.
- Repeat for i ranging from 1 to n
 - a. Set `result = result * i`.
- Return `result`.

Hand executing this algorithm demonstrates that if it receives the value 3, then it returns 6, and if it receives either 0 or 1, it returns 1. We can implement it as follows.

```
int computeFactorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

As we expect, when we call this method, we get the following results.

`computeFactorial(0)` → 1

`computeFactorial(1)` → 1

`computeFactorial(3)` → 6

The factorial method does not have to be defined and implemented this way. It can, in fact, be defined recursively. For example, consider the problem of computing the value of $5!$. We know that $4!$ and $5!$ can be written as follows.

$$4! = \underline{1 \times 2 \times 3 \times 4} = 24$$

$$5! = \underline{1 \times 2 \times 3 \times 4} \times 5 = 120$$

What if we notice that the value of $5!$ can be computed in terms of the value of $4!$. That is, the value $5!$ can be computed simply by multiplying the value of $4!$ by 5:

$$5! = 5 \times 4! = 5 \times 24 = 120$$

It turns out that this is a general pattern. The value of $5!$ can in turn be used to calculate $6!$,

$$6! = 6 \times 5! = 6 \times 120 = 720$$

and so on. Indeed, we can define the factorial function recursively, as follows.

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

To calculate $n!$ for any positive integer n , we need only know the value of $0!$,

$$0! = 1$$

and the fundamental relation between one factorial and the next:

$$n! = n \times (n-1)!$$

In general, an operation is said to be **defined recursively** if its definition consists of two parts:

- An **anchor** or **base case**, in which the value produced by the operation is specified for one or more values of the operands;
- An **inductive** or **recursive step**, in which the value produced for the current value of the operands is defined in terms of previously defined results and/or operand values.

For the factorial operation $!$ we have:

$$0! = 1 \quad \text{(the anchor or base case)}$$

$$\text{For } n > 0, n! = n \times (n-1)! \quad \text{(the inductive or recursive step)}$$

The first statement specifies a particular value produced by $!$, and the second statement defines the value produced for n in terms of value produced for $n-1$. We can specify this definition as a recursive algorithm as follows.³

Given:

- n is a non-negative integer.

Algorithm (for `computeFactorial()`):

1. Receive an integer n from the calling program.
2. If $n = 0$:
 - a. Return 1.

Otherwise:

- a. Return $n * \text{computeFactorial}(n-1)$.

We can implement this algorithm as follows.

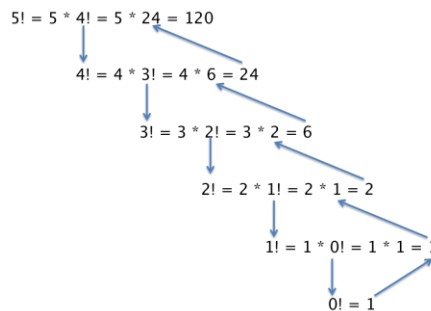
³ Note that a recursive definition with a slightly different anchor case can be given by observing that $0!$ and $1!$ are both 1. Although this alternative definition leads to a slightly more efficient implementation of `factorial()`, we will use the simpler definition in this introduction to recursion.


```

int computeFactorial(int n) {
    if (n == 0) {
        return 1; // the base case
    } else {
        return n * computeFactorial(n - 1); // the recursive case
    }
}

```

To see how this program works, consider using it to calculate 5!. The following diagram shows the computation as it recurses from the initial case down to the base case and then back up again.



Starting in the upper left, we must first calculate 4! because 5! is defined as the product of 5 and 4!. But to calculate 4! we must calculate 3! because 4! is defined as 4 * 3!. And to calculate 3!, we must apply the inductive step of the definition again, 3! = 3 * 2!, then again to find 2!, which is defined as 2! = 2 * 1!, and once again to find 1! = 1 * 0!. Since the value of 0! is given, we can now backtrack to find the value of 1!, then backtrack again to find the value of 2!, and so on, until we eventually obtain the value 120 for 5!.

As this example demonstrates, a recursive definition may require considerable bookkeeping to record information at the various levels of the recursion, because this information is used after the anchor case is reached to backtrack from one level to the preceding one. Fortunately, most modern high-level languages (including Processing) support recursion by automatically performing all of the necessary bookkeeping and backtracking.

Recall that our algorithm computes $n!$ under the assumption that n is non-negative. This assumption is important. To see the reason for this, consider what would happen if `computeFactorial()` were called with a negative integer, as in

```
computeFactorial(-1);
```

Since -1 is not equal to 0, the inductive step

```

else
    return n * computeFactorial(n-1);

```

would be performed, recursively calling `computeFactorial(-2)`. Execution of this call would begin, and since -2 is not equal to 0, the inductive step

```
else
    return n * computeFactorial(n-1);
```

would be performed, recursively calling `factorial(-3)`. It's clear at this point that this computation will never reach the base case, and would thus go on forever. When executed, this behavior would continue until memory was exhausted, at which point the program would terminate abnormally, possibly producing an error message like

```
Stack overruns Heap.
```

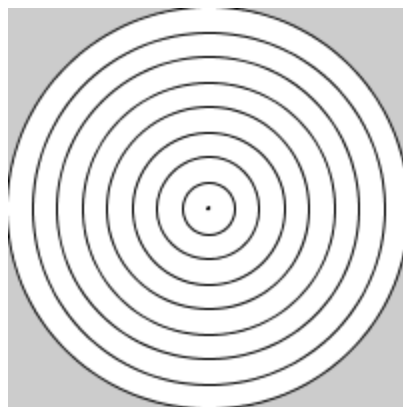
Such behavior is described as *infinite recursion* and is obviously undesirable. To avoid it we would have to program defensively by including the parameter-validity check such as the following.

```
int computeFactorial(int n) {
    if (n < 0) {
        println("Error: n! is not defined for negative n.");
    } else if (n == 0) {
        return 1;
    } else {
        return n * computeFactorial(n - 1);
    }
}
```

This improved version of the method refuses to start any infinite recursion when given a negative `n`.⁴

7.6.2. Drawing Fractal Circles

We are now ready to compute the recursive circles example shown at the beginning of this section. We'll start with a simpler case of concentric circles shown here. These circles have diameters that decrease by 25 screen pixels on each repetition. The smallest circle, with diameter 1, appears as a point in the middle.

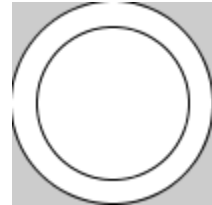
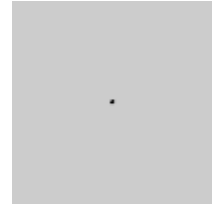


To design a recursive algorithm for any problem, we need identify the base and the recursive cases. Indeed, if we can't do this, then we must give up on the recursive approach entirely. The trick is to

⁴ We will discuss a more principled approach to error handling in a later chapter.

discern and then specify both the base and the recursive cases. In the case of concentric circles, we have the following cases:

- **Base case** – To see the base case in our concentric circles, we need to peer deeply into the structure of the diagram and notice that at the lowest level, when the circle diameter is set at only 1 pixel, we are drawing a single, small circle of diameter 1.
- **Recursive case** – To see the recursive case, we can look at the highest level, when the circle diameter is set to 25 pixels less than the width of the display window. In this case, in addition to drawing the large circle, we must also draw a smaller circle centered on the same point but with a diameter that is 10 pixels smaller.



Given these two cases, we can specify a recursive algorithm for a `drawCircle()` method as follows.

Given:

- `WIDTH` is the width of the display window.
- The height of display window equals its width.
- We have set `LIMIT` to 1, the smallest circle we want to draw

Algorithm (for `drawCircles()`):

1. Receive integers `x`, `y` and `diameter` from the calling program, representing the (`x` `y`) coordinates and diameter of the desired circle respectively.
2. If `diameter` \leq `LIMIT`:
 - a. Draw a circle at position (`x`, `y`) of diameter `diameter`.Otherwise:
 - a. Draw a circle at position (`x`, `y`) of diameter `diameter`.
 - b. Call `drawCircles(x, y, diameter - 25)`.

In this algorithm, step 2.a handles the base case and steps a and b of the Otherwise part of step 2 handle the recursive case. We notice at this point that the algorithm repeats the draw circle operation (steps 2.a and 2.Otherwise.a) and realize that we can simplify step 2 as follows.

1. Receive integers `x`, `y` and `diameter` from the calling program, representing the (`x` `y`) coordinates and diameter of the desired circle respectively.
2. Draw a circle at position (`x`, `y`) of diameter `diameter`.
3. If `diameter` $>$ `LIMIT`:
 - a. Call `drawCircles(x, y, diameter - 25)`.

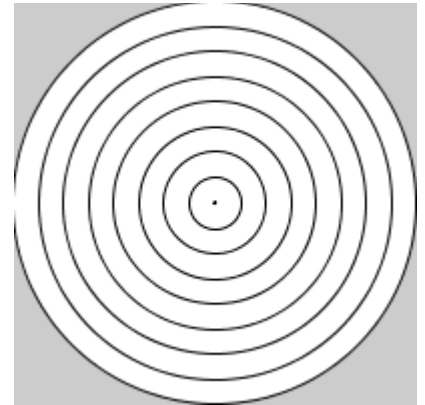
In this modified algorithm, step 2 draws the circle required by both the base and recursive cases, and step 3 inverts the selection condition and handles only the additional recursive call required

by the recursive step. This equivalent but slightly more elegant algorithm can be implemented as follows.

```
final int WIDTH = 201, HEIGHT = WIDTH;
final int LIMIT = 1;

void setup() {
  size(WIDTH, HEIGHT);
  smooth();
  drawCircles(WIDTH/2, HEIGHT/2, WIDTH);
}

void drawCircles(int x, int y, int diameter) {
  ellipse(x, y, diameter, diameter);
  if (diameter > LIMIT) {
    drawCircles(x, y, diameter - 25);
  }
}
```



We can now make a small modification to this concentric circles algorithm to create the fractal circles example given at the beginning of this section. The algorithm is the same except that the recursive case draws two smaller circles, each with half the diameter of the outer circle, one on the left and one on the right.

Given:

- WIDTH is the width of the display window.
- The height of display window equals its width.
- We have set LIMIT to 1, the smallest circle we want to draw

Algorithm (for drawCircles()):

3. Receive integers x , y and $diameter$ from the calling program, representing the (x, y) coordinates and diameter of the desired circle respectively.
4. If $diameter \leq LIMIT$:
 - a. Draw a circle at position (x, y) of diameter $diameter$.

Otherwise:

- a. Draw a circle at position (x, y) of diameter $diameter$.
- b. Call drawCircles($x - diameter/4, y, diameter/2$).**
- c. Call drawCircles($x + diameter/4, y, diameter/2$).**

We can implement this algorithm as follows.

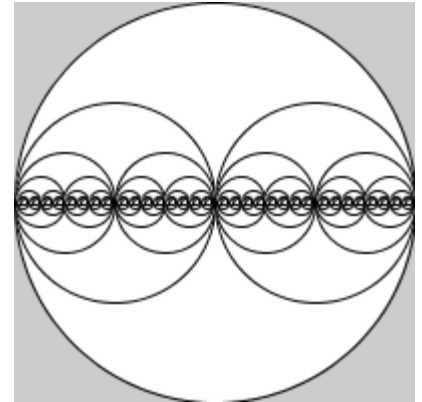
```

final int WIDTH = 201, HEIGHT = WIDTH;
final int LIMIT = 1;

void setup() {
  size(WIDTH, HEIGHT);
  smooth();
  drawCircles(WIDTH/2, HEIGHT/2, WIDTH);
}

void drawCircles(int x, int y, int diameter) {
  ellipse(x, y, diameter, diameter);
  if (diameter > LIMIT) {
    drawCircles(x-diameter/4, y, diameter/2);
    drawCircles(x+diameter/4, y, diameter/2);
  }
}

```



This code is surprisingly simple compared with the apparent complexity of the output it produces. This demonstrates the power of recursion and helps to explain why fractals have proven to be such powerful characterizations of complex phenomena.