Control 2: Repetition

This unit focuses on controlling the flow of programs with iterative structures.

Syntax introduced:
`for`

The early history of computers is the history of automating calculation. A "computer" was originally a person who was paid to calculate math by hand. What we know as a computer today emerged from machines built to automate tedious mathematical calculations. The earliest mechanical computers were calculators developed for speed and accuracy in performing repetitive calculations. Because of this heritage, computers are excellent at executing repetitive tasks accurately and quickly. Modern computers are also logic machines. Building on the work of the logicians Leibniz and Boole, modern computers use logical operations such as AND, OR, and NOT to determine which lines of code are run and which are not.

Iteration

Iterative structures are used to compact lengthy lines of repetitive code. Decreasing the length of the code can make programs easier to manage and can also help to reduce errors. The table below shows equivalent programs written without an iterative structure and with a `for` structure. The original 14 lines of code on the left are reduced to the 4 lines on the right:

Original code

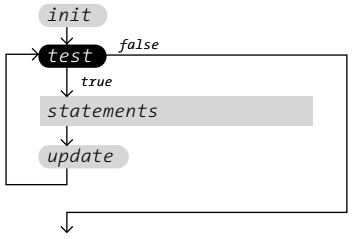
```
size(200, 200);
line(20, 20, 20, 180);
line(30, 20, 30, 180);
line(40, 20, 40, 180);
line(50, 20, 50, 180);
line(60, 20, 60, 180);
line(70, 20, 70, 180);
line(80, 20, 80, 180);
line(90, 20, 90, 180);
line(100, 20, 100, 180);
line(110, 20, 110, 180);
line(120, 20, 120, 180);
line(130, 20, 130, 180);
line(140, 20, 140, 180);
```

Code expressed using a `for` structure

```
size(200, 200);
for (int i = 20; i < 150; i += 10) {
  line(i, 20, i, 180);
}
```

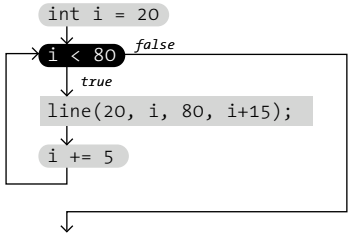
General case `for` structure

```
for (init; test; update) {
  statements
}
```



A specific `for` structure

```
for (int i = 20; i < 80; i += 5) {
  line(20, i, 80, i+15);
}
```



Repetition

The flow of a `for` structure shown as a diagram. These images show the central importance of the test statement in deciding whether to run the code in the block or to exit. The general case shows the generic format, and the specific case shows one example of how the format can be used within a program.

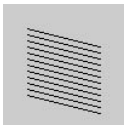
The `for` structure performs repetitive calculations and is structured like this:

```
for (init; test; update) {
  statements
}
```

The parenthesis associated with the structure enclose three statements: *init*, *test*, and *update*. The statements inside the block are run continuously while the test evaluates to `true`. The *init* portion assigns the initial value of the variable used in the test. The *update* is used to modify the variable after each iteration through the block. A `for` structure runs in the following sequence:

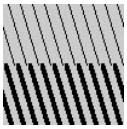
1. The *init* statement is run
2. The *test* is evaluated to *`true`* or *`false`*
3. If the *test* is *`true`*, continue to step 4. If the *test* is *`false`*, jump to step 6
4. Run the statements within the block
5. Run the *update* statement and jump to step 2
6. Exit the structure and continue running the program

The following examples demonstrate how the `for` structure is used within a program to control the way shapes are drawn to the display window.



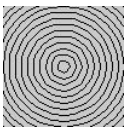
6-01

```
// The init is "int i = 20", the test is "i < 80",
// and the update is "i += 5". Notice the semicolon
// terminating the first two elements
for (int i = 20; i < 80; i += 5) {
  // This line will continue to run until "i"
  // is greater than or equal to 80
  line(20, i, 80, i+15);
}
```



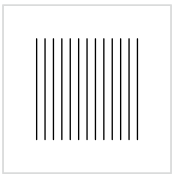
6-02

```
for (int x = -16; x < 100; x += 10) {
  line(x, 0, x+15, 50);
}
strokeWeight(4);
for (int x = -8; x < 100; x += 10) {
  line(x, 50, x+15, 100);
}
```

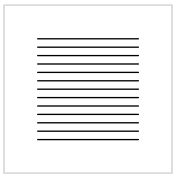


6-03

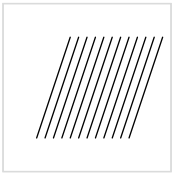
```
noFill();
for (int d = 150; d > 0; d -= 10) {
  ellipse(50, 50, d, d);
}
```



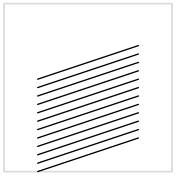
```
for (int x = 20; x <= 80; x += 5) {
  line(x, 20, x, 80);
}
```



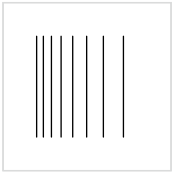
```
for (int x = 20; x <= 80; x += 5) {
  line(20, x, 80, x);
}
```



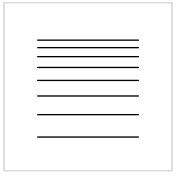
```
for (int x = 20; x < 80; x += 5) {
  line(x+20, 20, x, 80);
}
```



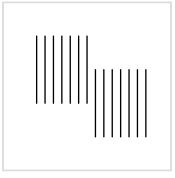
```
for (float x = 80; x > 20; x -= 5) {
  line(20, x+20, 80, x);
}
```



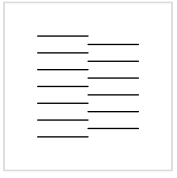
```
for (float x = 20; x < 80; x *= 1.2) {
  line(x, 20, x, 80);
}
```



```
for (float x = 80; x > 20; x /= 1.2) {
  line(20, x, 80, x);
}
```



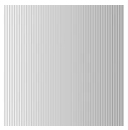
```
for (int x = 20; x <= 85; x += 5) {
  if (x <= 50) {
    line(x, 20, x, 60);
  } else {
    line(x, 40, x, 80);
  }
}
```



```
for (int x = 20; x <= 80; x += 5) {
  if ((x % 10) == 0) {
    line(20, x, 50, x);
  } else {
    line(50, x, 80, x);
  }
}
```

All for one and one for all

The `for` *structure is flexible, but it always follows the rules. These examples show how it can be used to generate various patterns.*



```
for (int i = 0; i < 100; i += 2) {
  stroke(255-i);
  line(i, 0, i, 200);
}
```

6-04

Nested iteration

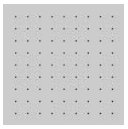
The `for` structure produces repetitions in one dimension. Nesting one of these structures into another compounds their effect, creating iteration in two dimensions. Instead of drawing 9 points and then drawing another 9 points, they combine to create 81 points; for each point drawn in the outer structure, 9 points are drawn in the inner structure. The inner structure runs through a complete cycle for each single iteration of the outer structure. In the following examples, the two dimensions are translated into x-coordinates and y-coordinates:

```
for (int y = 10; y < 100; y += 10) {
  point(10, y);
}
```

6-05


```
for (int x = 10; x < 100; x += 10) {
  point(x, 10);
}
```

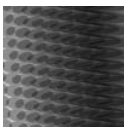
6-06



```
for (int y = 10; y < 100; y += 10) {
  for (int x = 10; x < 100; x += 10) {
    point(x, y);
  }
}
```

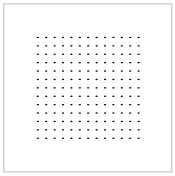
6-07

This technique is useful for creating diverse patterns and effects. The numbers produced by embedding iterative elements can be applied to color, position, size, transparency, and any other visual attribute.

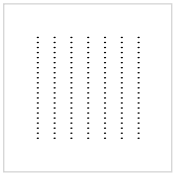


```
fill(0, 76);
noStroke();
smooth();
for (int y = -10; y <= 100; y += 10) {
  for (int x = -10; x <= 100; x += 10) {
    ellipse(x + y/8.0, y + x/8.0, 15 + x/2, 10);
  }
}
```

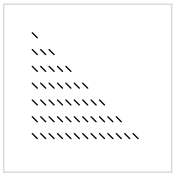
6-08



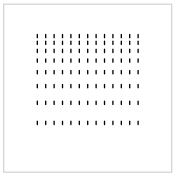
```
for (int y = 20; y <= 80; y += 5) {
  for (int x = 20; x <= 80; x += 5) {
    point(x, y);
  }
}
```



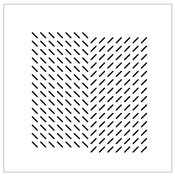
```
for (int y = 20; y <= 80; y += 3) {
  for (int x = 20; x <= 80; x += 10) {
    point(x, y);
  }
}
```



```
for (int y = 20; y <= 80; y += 10) {
  for (int x = 20; x <= y; x += 5) {
    line(x, y, x-3, y-3);
  }
}
```



```
for (float y = 20; y <= 80; y *= 1.2) {
  for (int x = 20; x <= 80; x += 5) {
    line(x, y, x, y-2);
  }
}
```

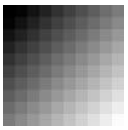


```
for (int y = 20; y <= 85; y += 5) {
  for (int x = 20; x <= 85; x += 5) {
    if (x <= 50) {
      line(x, y, x-3, y-3);
    } else {
      line(x, y, x-3, y+3);
    }
  }
}
```

```
for (int y = 20; y <= 80; y += 5) {
  for (int x = 20; x <= 80; x += 5) {
    if ((x % 10) == 0) {
      line(x, y, x+3, y-3);
    } else {
      line(x, y, x+3, y+3);
    }
  }
}
```

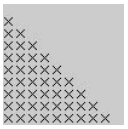
Embedding (nesting)

Embedding one `for` structure inside another is a highly malleable technique for drawing patterns. These examples show only a few of the possible options.



```
noStroke();
for (int y = 0; y < 100; y += 10) {
  for (int x = 0; x < 100; x += 10) {
    fill((x+y) * 1.4);
    rect(x, y, 10, 10);
  }
}
```

6-09



```
for (int y = 1; y < 100; y += 10) {
  for (int x = 1; x < y; x += 10) {
    line(x, y, x+6, y+6);
    line(x+6, y, x, y+6);
  }
}
```

6-10

Formatting code blocks

It's important to space code so the blocks are clear. The lines inside a block are typically offset to the right with spaces or tabs. When programs become longer, clearly defining the beginning and end of the block reveals the structure of the program and makes it more legible. This is the convention used in this book:

```
int x = 50;

if (x > 100) {
  line(20, 20, 80, 80);
} else {
  line(80, 20, 20, 80);
}
```

6-11

This is an alternative format that is sometimes used elsewhere:

```
int x = 50;

if (x > 100)
{
  line(20, 20, 80, 80);
}
else
{
  line(20, 80, 80, 20);
}
```

6-12

It's essential to use formatting to show the hierarchy of your code. The Processing environment will attempt basic formatting as you type, and you can use the "Auto Format" function from the Tools menu to clean your code at any time. The `line()` function in the following code fragment is inside the `if` structure, but the spacing does not reveal this at a quick glance. Avoid formatting code like this:

6-13

```
int x = 50;

if (x > 100) {
line(20, 20, 80, 80);  // Avoid formatting code like this
} else {               // because it makes it difficult to see
line(80, 20, 20, 80);  // what is inside the block
}
```

Exercises

1. *Draw a regular pattern with five lines. Rewrite the code using a `for` structure.*
2. *Draw a dense pattern by embedding two `for` structures.*
3. *Combine two relational expressions with a logical operator to control the form of a pattern.*