Control 1: Decisions

This unit focuses on controlling the flow of a program with conditional structures. Logical operators for extending relational expressions are introduced.

Syntax introduced:

```
> (greater than), < (less than)
>= (greater than or equal to), <= (less than or equal to)
== (equality), != (inequality)
if, else, {} (braces)
|| (logical OR), && (logical AND), ! (logical NOT)
```

The programs we've seen so far run each line of code in sequence. They run the first line, then the second, then the third, etc. The program stops when the last line is run. It's often beneficial to change this order—sometimes skipping lines or repeating lines many times to perform a repetitive action. Although the lines of code that comprise a program are always positioned in an order from top to bottom on the page, this doesn't necessarily define the order in which each line is run. This order is called the *flow* of the program. Flow can be changed by adding elements of code called control structures.

Relational expressions

What is truth? It's easy to answer this difficult philosophical question in the context of programming because the logical notions of true and false are well defined. Code elements called relational expressions evaluate to `true` and `false`. A relational expression is made up of two values that are compared with a relational operator. In Processing, two values can be compared with relational operators as follows:

Expression	Evaluation
`3 > 5`	`false`
`3 < 5`	`true`
`5 < 3`	`false`
`5 > 3`	`true`

Each of these statements can be converted to English. Using the first row as an example, we can say, "Is three greater than five?" The answer "no" is expressed with the value `false`. The next row can be converted to "Is three less than five?" The answer is "yes" and is expressed with the value `true`. A relational expression, two values compared with a relational operator, evaluates to `true` or `false`—there are no other possible values. The relational operators are defined as follows:

Operator	Meaning
`>`	`greater than`
`<`	`less than`
`>=`	`greater than or equal to`
`<=`	`less than or equal to`
`==`	`equivalent to`
`!=`	`not equivalent to`

The following lines of code show the results of comparing the same group of numbers with different relational operators:

5-01

```
println(3 > 5);   // Prints "false"
println(5 > 3);   // Prints "true"
println(5 > 5);   // Prints "false"

println(3 < 5);   // Prints "true"
println(5 < 3);   // Prints "false"
println(5 < 5);   // Prints "false"

println(3 >= 5);  // Prints "false"
println(5 >= 3);  // Prints "true"
println(5 >= 5);  // Prints "true"

println(3 <= 5);  // Prints "true"
println(5 <= 3);  // Prints "false"
println(5 <= 5);  // Prints "true"
```

The equality operator, the == symbol, determines whether two values are equivalent. It is different from the = symbol, which assigns a value, but the two are often used erroneously in place of each other. The only way to avoid this mistake is to be careful. It's similar to using "their" instead of "there" when writing in English—a mistake that even experienced writers sometimes make. The != symbol is the opposite of == and determines whether two values are not equivalent.

5-02

```
println(3 == 5);  // Prints "false"
println(5 == 3);  // Prints "false"
println(5 == 5);  // Prints "true"

println(3 != 5);  // Prints "true"
println(5 != 3);  // Prints "true"
println(5 != 5);  // Prints "false"
```

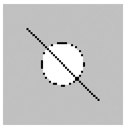
Conditionals

Conditionals allow a program to make decisions about which lines of code run and which do not. They let actions take place only when a specific condition is met. Conditionals allow a program to behave differently depending on the values of their variables. For example, the program may draw a line or an ellipse depending on the value of a variable. The `if` structure is used in Processing to make these decisions:

```
if (test) {
  statements
}
```

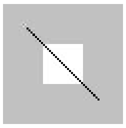
The test must be an expression that resolves to `true` or `false`. When the test expression evaluates to `true`, the code inside the { (left brace) and } (right brace) is run. If the expression is `false`, the code is ignored. Code inside a set of braces is called a block.

The following three examples present the same code with different values for the `x` variable. Because this variable is used in the test for the `if` structure, changing it affects which lines of code are run. Changing the value causes an ellipse, rectangle, or neither to draw to the display window.



5-03

```
// The text expressions are "x > 100" and "x < 100"
// Because x is 150, the code inside the first block
// runs and the ellipse draws, but the code in the second
// block is not run and the rectangle is not drawn
int x = 150;
if (x > 100) {              // If x is greater than 100,
  ellipse(50, 50, 36, 36);  // draw this ellipse
}
if (x < 100) {              // If x is less than 100
  rect(35, 35, 30, 30);     // draw this rectangle
}
line(20, 20, 80, 80);
```

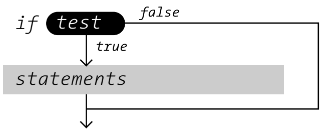


5-04

```
// Because x is 50, only the rectangle draws
int x = 50;
if (x > 100) {              // If x is greater than 100,
  ellipse(50, 50, 36, 36);  // draw this ellipse
}
if (x < 100) {              // If x is less than 100,
  rect(33, 33, 34, 34);     // draw this rectangle
}
line(20, 20, 80, 80);
```

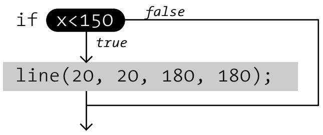
General case `if` structure

```
if (test) {
  statements
}
```



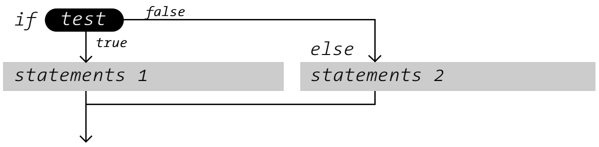
A specific `if` structure

```
if (x < 150) {
  line(20, 20, 180, 180);
}
```



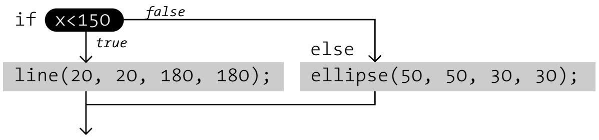
General case `if`/`else` structure

```
if (test) {
  statements 1
} else {
  statements 2
}
```



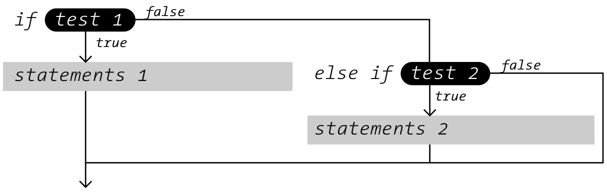
A specific `if`/`else` structure

```
if (x < 150) {
  line(20, 20, 180, 180);
} else {
  ellipse(50, 50, 30, 30);
}
```



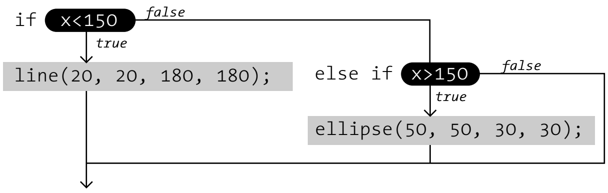
General case `if`/`else if` structure

```
if (test 1) {
  statements 1
} else if (test 2) {
  statements 2
}
```

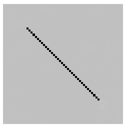


A specific `if`/`else if` structure

```
if (x < 150) {
  line(20, 20, 180, 180);
} else if (x > 150) {
  ellipse(50, 50, 30, 30);
}
```



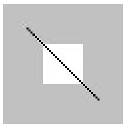
Decisions
The flow of an `if`, `else`, and `else if` structure shown as a diagram. The code inside each block is run if the test evaluates to true. For each set of diagrams, the general case shows the generic format and the specific case shows one example of how the format can be used within a program.



5-05

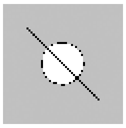
```
// Because x is 100, only the line draws
int x = 100;
if (x > 100) {              // If x is greater than 100,
  ellipse(50, 50, 36, 36);  // draw this ellipse
}
if (x < 100) {              // If x is less than 100,
  rect(33, 33, 34, 34);     // draw this rectangle
}
line(20, 20, 80, 80);       // Always draw the line
```

To run a different set of code when the relational expression for an `if` structure is not `true`, use the `else` keyword. The keyword `else` extends an `if` structure so that when the expression associated with the structure is `false`, the code in the `else` block is run instead.



5-06

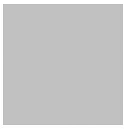
```
//  Because x is 90, only the rectangle draws
int x = 90;
if (x > 100) {              // If x is greater than 100,
  ellipse(50, 50, 36, 36);  // draw this ellipse
} else {                    // Otherwise,
  rect(33, 33, 34, 34);     // draw this rectangle
}
line(20, 20, 80, 80);       // Always draw the line
```



5-07

```
// Because x is 290, only the ellipse draws
int x = 290;
if (x > 100) {              // If x is greater than 100,
  ellipse(50, 50, 36, 36);  // draw this ellipse
} else {                    // Otherwise,
  rect(33, 33, 34, 34);     // draw this rectangle
}
line(20, 20, 80, 80);       // Always draw the line
```

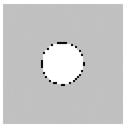
Conditionals can be embedded within other conditionals to control which lines of code will run. In the next example, the code for drawing the ellipse or line can be reached only if `x` is larger than 100. If this expression evaluates to `true`, a second comparison of `x` determines which of these shapes will be drawn.



```
// If x is greater than 100 and less than 300, draw the
// ellipse. If x is greater than or equal to 300, draw
// the line. If x is not greater than 100, draw the
// rectangle. Because x is 420, only the line draws.
int x = 420;
if (x > 100) {    // First test to draw ellipse or line
  if (x < 300) {  // Second test determines which to draw
    ellipse(50, 50, 36, 36);
  } else {
    line(50, 0, 50, 100);
  }
} else {
  rect(33, 33, 34, 34);
}
```

5-08

Conditionals can be extended further by combining an `else` with an `if`. This allows conditionals to use multiple tests to determine which lines the program should run. This technique is used when there are many choices and only one can be selected at a time.



```
// If x is less than or equal to 100, then draw
// the rectangle. Otherwise, if x is greater than
// or equal to 300, draw the line. If x is between
// 100 and 300, draw the ellipse. Because x is 101,
// only the ellipse draws.
int x = 101;
if (x <= 100) {
  rect(33, 33, 34, 34);
} else if (x >= 300) {
  line(50, 0, 50, 100);
} else {
  ellipse(50, 50, 36, 36);
}
```

5-09

Logical operators

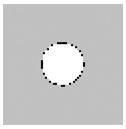
Logical operators are used to combine two or more relational expressions and to invert logical values. They allow for more than one condition to be considered simultaneously. The logical operators are symbols for the logical concepts of AND, OR, and NOT:

Operator	Meaning
`&&`	AND
`||`	OR
`!`	NOT

The following table outlines all possible combinations and the results.

Expression	Evaluation
`true && true`	`true`
`true && false`	`false`
`false && false`	`false`
`true || true`	`true`
`true || false`	`true`
`false || false`	`false`
`!true`	`false`
`!false`	`true`

The logical OR operator, two vertical bars (sometimes called pipes), makes the relational expression `true` if only one part is `true`. The following example shows how to use it:



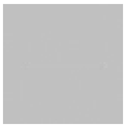
5-10

```
int a = 10;
int b = 20;
// The expression "a > 5" must be true OR "b < 30"
// must be true. Because they are both true, the code
// in the block will run.
if ((a > 5) || (b < 30)) {
  line(20, 50, 80, 50);
}
// The expression "a > 15" is false, but "b < 30"
// is true. Because the OR operator requires only one part
// to be true in the entire expression, the code in the
// block will run.
if ((a > 15) || (b < 30)) {
  ellipse(50, 50, 36, 36);
}
```

Compound logical expressions can be tricky to figure out, but they are simpler when looked at step by step. Parentheses are useful hints in determining the order of evaluation. Looking at the test of the `if` structure in line 6 of the previous example, first the variables are replaced with their values, then each subexpression is evaluated, and finally the expression with the logical operator is evaluated:

```
Step 1    (a > 5) || (b < 30)
Step 2    (10 > 5) || (20 < 30)
Step 3    true || true
Step 4    true
```

The logical AND operator, two ampersands, allows the entire relational statement to be `true` only if both parts are `true`. The following example is the same as the last except the logical OR operators have been changed to the logical AND. Because each operator compares the values differently, only the line is drawn here, whereas the previous example drew both the line and circle.



5-11

```
int a = 10;
int b = 20;
// The expression "a > 5" must be true AND "b < 30"
// must be true. Because they are both true, the code
// in the block will run.
if ((a > 5) && (b < 30)) {
  line(20, 50, 80, 50);
}
// The expression "a > 15" is false, but "b < 30" is
// true. Because the AND operator requires both to be
// true, the code in the block will not run.
if ((a > 15) && (b < 30)) {
  ellipse(50, 50, 36, 36);
}
```

Technically, the steps shown above aren't the whole story. When using AND, the first part of the expression will be evaluated. If that part is `false`, then the second part of the expression won't even be evaluated. For example, in this expression . . .

```
(a > 5) && (b < 30)
```

. . . if `a > 5` evaluates to `false`, then `b < 30` is ignored for efficiency. This is called a *short circuit* operator. The same happens for the OR operator, where the first `true` statement will end evaluation. For example, if the expression is:

```
(a > 5) || (b < 30)
```

If `a > 5` is true, then the `b < 30` will be ignored, because the entire expression will evaluate to true, regardless of the value of `b < 30`. Outside of efficiency, this has many practical applications in more advanced code.

The logical NOT operator is an exclamation mark. It inverts the logical value of the associated `boolean` variables. It changes `true` to `false`, and `false` to `true`. The logical NOT operator can be applied only to `boolean` variables.

5-12

```
boolean b = true;   // Assign true to b
println(b);         // Prints "true"
println(!b);        // Prints "false"
b = !b;             // Assign false to b
println(b);         // Prints "false"
println(!b);        // Prints "true"
println(5 > 3);     // Prints "true"
println(!(5 > 3));  // Prints "false"
int x = 5;
println(!x);  // ERROR! It's only possible to ! a boolean variable
```

5-13

```
// Because b is true, the line draws
boolean b = true;
if (b == true) {            // If b is true,
  line(20, 50, 80, 50);     // draw the line
}
if (!b == true) {           // If b is false,
  ellipse(50, 50, 36, 36);  // draw the ellipse
}
```

Exercises

1. *Create a few relational expressions and print their evaluation to the console with* `println()`*.*
2. *Create a composition with a series of lines and ellipses. Use an* `if` *structure to select which lines of code to run and which to skip.*
3. *Add an* `else` *to the code from exercise 2 to change which code is run.*