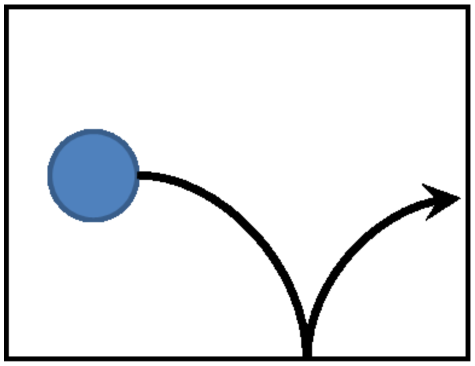
Chapter 5. Selection

In Chapter 1 we saw that algorithms deploy sequence, selection and repetition statements in combination to specify computations. Since that time, however, the computations that we've implemented have been strictly sequential in nature, starting with the first statement and proceeding sequentially through to the last statement. Many interesting and useful computations require *selective* execution, in which one path of execution is selected from many potential paths.

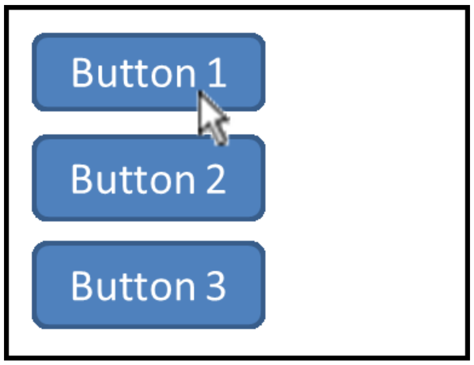
This chapter introduces the use of selection statements in Processing. Repetition statements are deferred until Chapter 7.

5.1. Examples: Going beyond Sequence

To this point in the text, the programs we have designed and implemented have been sequential in nature. We found this flow of control useful in achieving a number of interesting goals, but to be honest, we were significantly constrained in what we could attempt.



Consider, for example, the falling ball that we implemented in Chapter 4, which modeled a moving ball that appears to fall to the bottom of the screen with gravitational acceleration. While interesting, our solution was less than satisfying given that the ball flies off the output pane and never "bounces" back up when it hits the floor. We'd like the ball to bounce, as shown here. Modeling a ball that bounces like this requires that the program reverse the ball's direction of motion at the precise point when the ball reaches the bottom of the pane. The program must be selective about when it does this, so simple sequential execution is not sufficient.



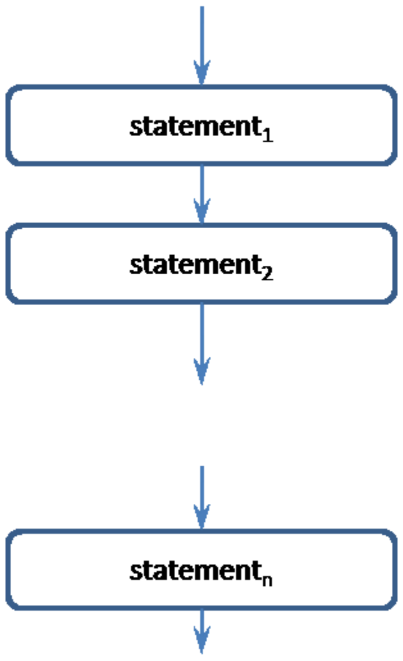
Similarly, consider the interactive programs that we implemented in Chapter 4. It was nice to place objects on the output pane using mouse clicks, but there are times when we'd like to be able to initiate actions only when the mouse press is in a certain location. Implementing button pressing interfaces, as shown here, require this selective behavior. The action is initiated only if the mouse press is "on" the button; other actions are initiated if the press is on another button.

In this chapter we will introduce the selection statements provided by Processing to perform these two sorts of selective tasks.

5.1.1. Sequence

This sequential flow of control can be visualized as shown in the diagram on the right. Programs execute their statements one at a time, from top to bottom. This is implemented in Processing using statement blocks as shown here:

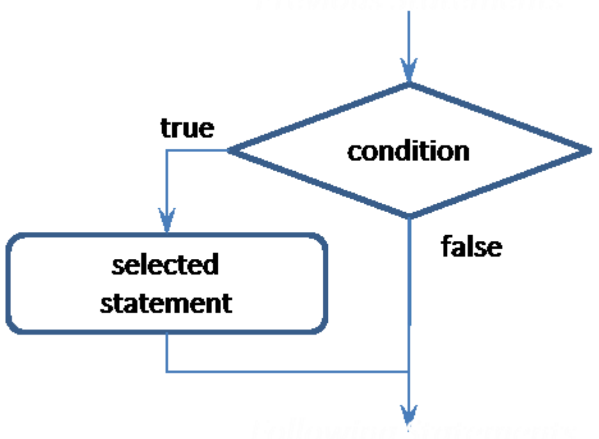
```
{
    statement1;
    statement2;
    …
    statementn;
}
```



In Chapter 4, we introduced methods, which re-route the flow of control temporarily, first from the calling program to the method and then back again, but the fundamental flow of control is still sequential. Methods help us implement abstraction, but do not help us implement selective behavior.

5.1.2. Selection

Selective execution allows programs to select their behavior based on an appropriate condition and can be visualized as shown in the diagram to the right. Execution flows from the top to the logical condition. If the expression evaluates to true, then the selected statement is executed, otherwise control flows past the statement without executing it. This simple selection mechanism is powerful enough to implement the bouncing balls and interactive buttons described above.



Processing implements selective behavior using the **`if`** statement.

Diagrams of this sort (and the sequential diagram shown above) are called *flow charts* because they graphically depict the flow of control through a program. Statements are represented as boxes. Selection conditions are represented as diamonds, which indicates that the specified condition must be checked. If the condition is true, control follows the arrow labeled "true" and executes the operation shown in the box on the left. Otherwise, control follows the arrow labeled "false".

5.2. Using the `if` Statement

Consider the task of determining an appropriate output message based on the value of a variable containing a user's guess in a simple guessing game. The correct answer is 3, so if the user guesses 3, then we would like to print a suitably congratulatory message. In previous chapters, we might have written a simple statement like the following:

```
println(‘‘You win!’’);
```

Unfortunately, this statement would be executed in all cases, that is, un-selectively. Instead, we'd like to implement the behavior specified in the following algorithm.

Given:

- `guess` is set to an integer representing the user's guess.

Algorithm:

```
1. If  guess equals 3:
    a. Print a congratulatory message;
```

To implement this algorithm in Processing, we use the following statement:

```
if (guess == 3) {
  println("You win!");
}
```

This program segment assumes that the variable `guess` has been assigned some integer value. It contains a single `if` statement that specifies a Boolean condition (`guess == 3`). Recall that we discussed Boolean expressions in some detail in Chapter 3. If this condition evaluates to true, then Processing executes the statement block immediately following the condition. In this example, if the value of **`guess`** is 3, then the string "You win!" is printed on the console; otherwise the statement block is skipped entirely and control proceeds to the following statements. Note that because the selectively-executed statement is implemented as a block, we could include multiple statements to be selected.

The **if** statement has the following general pattern:

if Statement Pattern

```
if (condition) {
   statements
}
```

or

```
if (condition) {
   statements1
} else {
   statements2
}
```

- *condition* is a Boolean expression indicating which branch of the statements to follow;
- *statements$_i$* are the sets of statements executed based on the conditions.

5.2.1. Adding an else clause

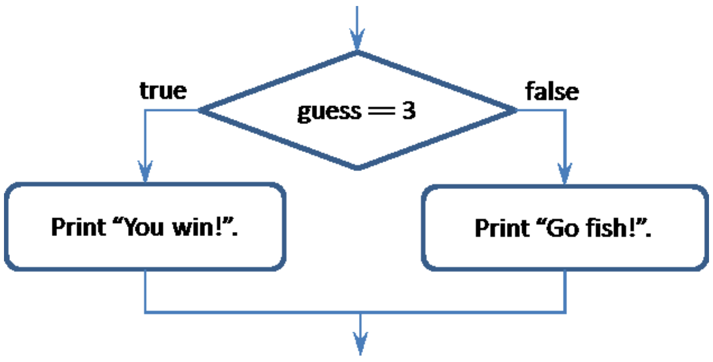
The congratulatory program we've just wrote uses the first form of this **if** statement pattern. Now, consider the problem of extending our program to handle cases where the user guesses the wrong answer. For example, if the user doesn't guess 3, then we'd like to print a suitably discouraging message. We could specify this selective behavior by extending the previous algorithm as follows:

```
1. If guess equals 3:
      a. Print a congratulatory message;
   Otherwise:
      a. Print a discouraging message.
```

We can implement this algorithm in Processing using the following code segment:

```
if (guess == 3) {
  println("You win!");
} else {
  println("Go fish.");
}
```

In this revised code segment, if the condition evaluates to true, we get the behavior we saw before, but this time, if the condition evaluates to false, Processing executes the statement immediately following the else. The flow of control through this code can be visualized as shown in the flow chart to the right.



5.2.2. Revisiting the Bouncing Ball

To model a moving ball that bounces when it hits the bottom of the output pane, we can use an `if` statement with a Boolean expression designed to detect when the ball has "hit" the bottom of the output pane and a second `if` statement with a Boolean expression designed to detect when the ball has "hit" the side of the output pane. The bouncing algorithm would be as follows:

Given:

- The output frame is setup properly (see the falling ball example in Chapter 4).

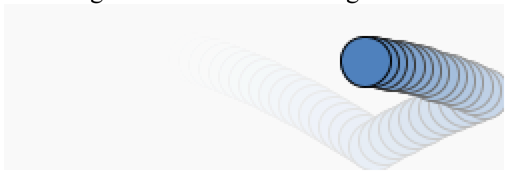
Algorithm:

1. Erase the ball from its previous location.
2. Draw the ball at its current location.
3. Move the ball to right and down.
4. If the ball hits the floor:
    a. Reverse its vertical movement.
5. If the ball hits either side:
    a. Reverse its horizontal movement.

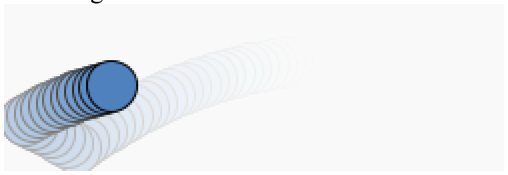
We've already implemented the first three steps of this algorithm in the falling ball example from Chapter 4, but we're now adding the selection statements in steps 4 and 5. These statements specify a bouncing behavior by reversing the velocity of the ball. When the ball hits the floor, the vertical component of the velocity is reversed; when the ball hits either wall, the horizontal component is reversed.

Note that neither of these selection statements require an otherwise clause; if the ball doesn't hit the floor or either of the walls, then the code continues to animate the unobstructed motion of the ball. Note also that these selection statements are independent of one another; the ball could either hit the floor or the wall, neither or both. The following code implements this extended "bouncing" ball algorithm.

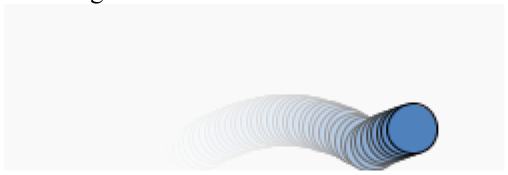
Bouncing off the floor and the right wall:



Bouncing off the floor and the left wall:



Bouncing twice off the floor:



```
final WIDTH = 250, HEIGHT = WIDTH/3;
final int BALL_SIZE = 25,
          BALL_RADIUS = BALL_SIZE / 2;

float ballX, ballY;     // ball location
float deltaX, deltaY;   // ball velocity
float gravity = 0.10;   // acceleration

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

  // Start the ball in the upper left.
  ballX = 15;
  ballY = 15;

  // Set the rate of change for x and y.
  deltaX = 5;
  deltaY = 0;
}

void draw() {
  // Erase the ball at the previous location.
  noStroke();
  fill(255, 30);
  rect(0, 0, width, height);

  // Draw the ball at the current location.
  stroke(0);
  fill(79, 129, 189);
  ellipse(ballX, ballY, BALL_SIZE, BALL_SIZE);

  // Move the ball to the right and down.
  ballX += deltaX;
  ballY += deltaY;
  deltaY += gravity;

  // Bounce the ball when it hits the floor.
  if ((ballY + BALL_RADIUS) >= height) {
    deltaY *= -0.85;
  }

  // Bounce the ball off the walls.
  if (((ballX + BALL_RADIUS) >= width) ||
      ((ballX - BALL_RADIUS) <= 0)) {
    deltaX *= -0.85;
  }
}
```

This code is largely copied from the example shown in Chapter 4 except for the following key changes: first, the beginning statements in the `draw()` method implement the trail that makes it easier to see the ball's motion in the output (in Chapter 4, we simply erased the previous location completely); second, and most importantly, the two `if` statements at the end of the `draw()` method model the bouncing behavior.

The first `if` statement checks to see if the ball is at the bottom of the output pane; this is true when the ball's y component, which marks the center of the ball, plus the ball's radius is beyond the height of the output pane. The code includes the radius because the ball should bounce based on its outer surface rather than its center. This models the ball bouncing up from the floor. The `if` statement has no `else` clause, which indicates that when the ball is not at the bottom of the output pane, it should continue to move as normal. To reverse the direction of the ball, the program multiplies `deltaY` by -0.85, which inverts the direction of change of `ballY`. The code uses -0.85 rather than -1.0 to model the effect of friction; if it had used -1.0, the ball would never stop bouncing.

The second `if` statement checks to see if the ball is either on the far right or the far left of the output pane. This models the ball bouncing off the walls. It does this using a more complex conditional expression than the previous examples use, checking whether the ball is beyond the right boundary or beyond the left boundary of the output pane.

5.3. Using Nested `if` Statements

Programs can place selection statements inside of one another. This is called *nesting*. Nested `if` statements can implement multi-branch selective behavior where the number of branches is greater than 2. Consider the task of assigning a grade letter grade given a percentage score. We can specify this behavior using the following algorithm:

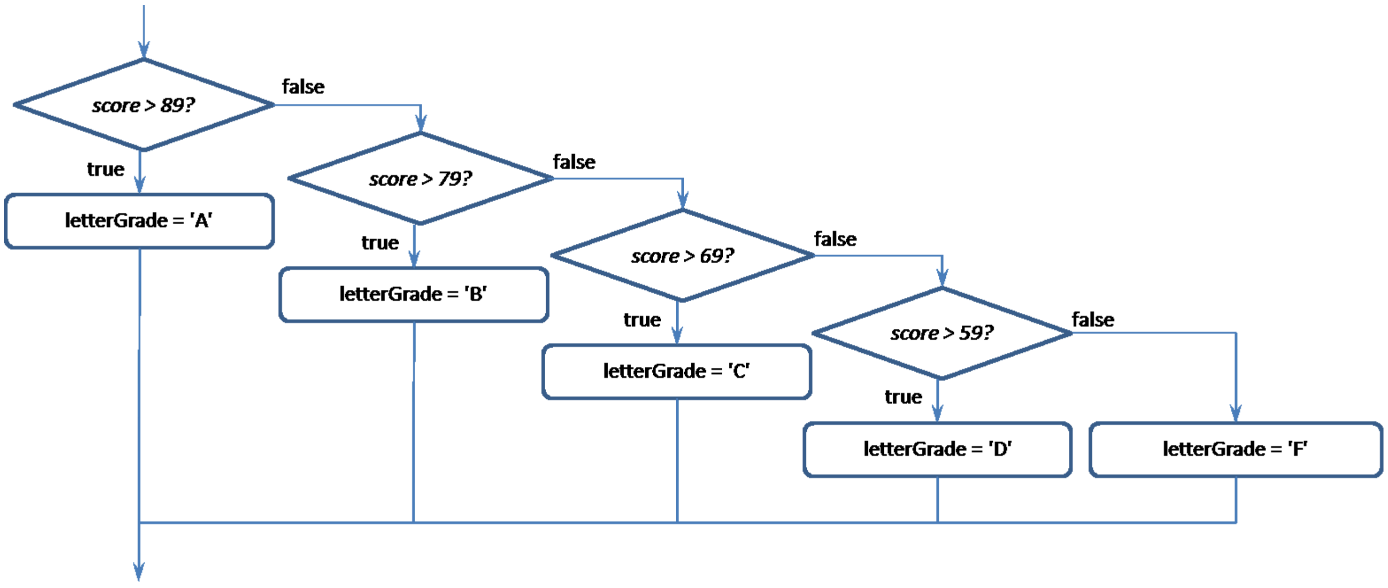
```
Given:
    • score is set to a valid percentage grade.
Algorithm:
    1. If the score is greater than 89:
        • Set the letterGrade to A.
       Otherwise, if the score is greater than 79:
        • Set the letterGrade to B.
       Otherwise, if the score is greater than 69:
        • Set the letterGrade to C.
       Otherwise, if the score is greater than 59:
        • Set the letterGrade to D.
       Otherwise
        • Set the letterGrade to F.
```

We can implement this algorithm with the following code segment:

```
if (score > 89) {
  letterGrade = 'A';
} else if (score > 79){
  letterGrade = 'B';
} else if (score > 69){
  letterGrade = 'C';
} else if (score > 59){
  letterGrade = 'D';
} else {
  letterGrade = 'F';
}
```

Notice that the order we specify the conditions matters: If the first condition is true (i.e., if the `score` is greater than 89), then `letterGrade` is set to A and the flow of control skips past all the other selection conditions. If the first condition is false, then the second condition is checked and if it is true (i.e., if `score` is greater than 79), then `letterGrade` is set to B and the flow of control skips the remaining conditions. Thus, `letterGrade` is set to B only if condition 1 is false and condition 2 is true. Conditions 3 and 4 are nested even lower. Finally, if the score hasn't met any of the previous conditions, then `letterGrade` is set to F. This "last resort" option has no selection condition and is thus executed when all of the previous conditions are false.

This control flow can be visualized as shown in the following flow chart:



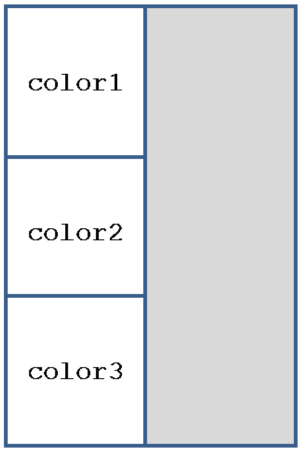
The nested **`if`** statement has the following general pattern:

Nested `if` Statement Pattern

```
if (condition₁) {
  statements₁
} else if (condition₂) {
  statements₂
}
  …
} else {
  statementsₙ
}
```

- `Conditionᵢ` is a Boolean expression indicating whether or not to execute

5.3.1. Revisiting the Interactive Buttons



As an example of the interactive button interface discussed above, consider the task of offering the user three color choices, as shown on the right. The idea is that when the user clicks the mouse on one of the choice "buttons" on the left of the sketch, then Processing will automatically set the chosen color area on the right to that chosen color. For example, if the user clicks color 1, then the output area on the right should be set to color 1. If the user clicks on the output color area, then the output area should revert to a default color.

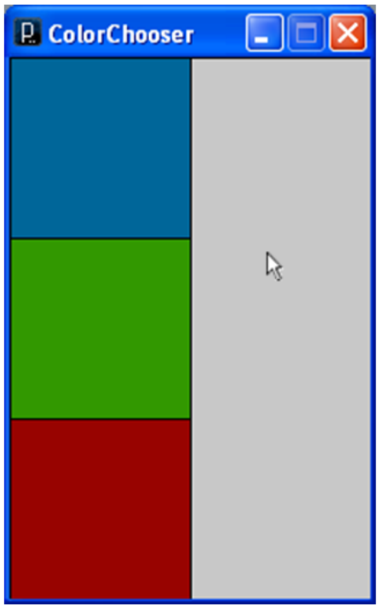
Because this task can have potentially many different color choices, far more than one or two, the nested-if statement is a natural choice for implementing the desired behavior.

We can specify this behavior using the following algorithm for the `mousePressed()` method.

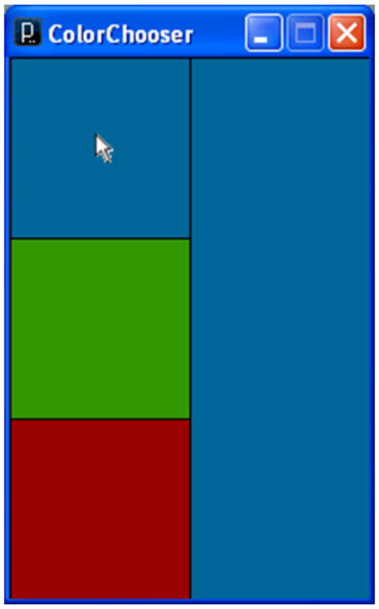
```
Given:
  • color1, color2, color3 and defaultColor are set to valid color values.
  • An output frame with three color-choice areas set up as shown in the output frame below.
  • The user has clicked the mouse.
Algorithm (for mousePressed()):
  1. If the user presses the mouse in the upper-left color choice area:
       • Set the output color to color1.
     Otherwise, if the user presses the mouse in the center color choice area:
       • Set the output color to color2.
     Otherwise, if the user presses the mouse in the lower-left color choice area:
       • Set the output color to color3.
     Otherwise:
       • Set the output color to a defaultColor.
```

This algorithm focuses only on the `mousePressed()` method, leaving the relatively simple `setup()` and `draw()` methods to the programmer's discretion. The flow of control through this algorithm thus assumes that the user has pressed the mouse somewhere on the output frame. The following program implements this algorithm.

The default output:



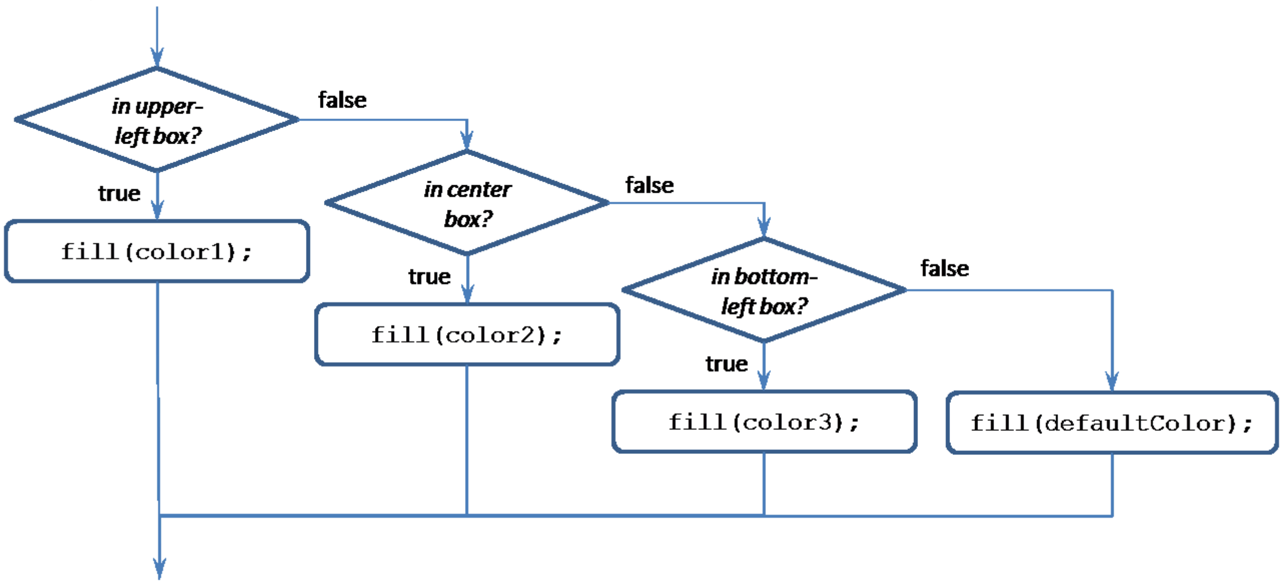
When the user chooses blue:



```
final int UNIT = 100;
final color COLOR1 = color(0, 102, 153),
            COLOR2 = color(50, 152, 0),
            COLOR3 = color(152, 3, 0),
            COLOR_DEFAULT = color(200);

void setup() {
  size(UNIT*2, UNIT*3);
  background(COLOR_DEFAULT);

  // Set up the three color choice boxes.
  fill(COLOR1);
  rect(0, 0, UNIT, UNIT);
  fill(COLOR2);
  rect(0, UNIT, UNIT, UNIT);
  fill(COLOR3);
  rect(0, UNIT*2, UNIT, UNIT);
  fill(COLOR_DEFAULT);
}

void draw() {
  // Re-draw the output color box.
  rect(UNIT, 0, UNIT*3, UNIT*3);
}

void mousePressed() {
  // The user has clicked the upper-left choice box.
  if ((mouseX > 0) && (mouseX < UNIT) &&
      (mouseY > 0) && (mouseY < UNIT)) {
    fill(COLOR1);

  // The user has clicked the center choice box.
  } else if ((mouseX > 0) && (mouseX < UNIT) &&
             (mouseY > UNIT) && (mouseY < UNIT*2)) {
    fill(COLOR2);

  // The user has clicked the lower-left choice box.
  } else if ((mouseX > 0) && (mouseX < UNIT) &&
             (mouseY > UNIT*2) && (mouseY < UNIT*3)) {
    fill(COLOR3);

  // The user has clicked in the output color box.
  } else {
    fill(COLOR_DEFAULT);
  }
}
```

This program responds to user mouse press events by determining which of the three color choice boxes they clicked on and then changes the color on the right to the chosen color. It includes simple implementations of `setup()`, which ensures that output frame is configured as required by the algorithm shown above, and `draw()`, which does the actual coloring of the output frame, but we focus on the implementation of `mousePressed()` here.

The code for the `mousePressed()` method can be visualized as shown in the following flow chart showing the nested if structure.



Note that the selection conditions in this flow of control are nested. If the first condition is true (i.e., if the user clicks in the upper-left box), then `color1` is used and the flow of control skips past all the other selection conditions. If the first condition is false, then the second condition is checked and if it is true (i.e., if the user has clicked in the center box), then `color2` is used and the flow of control skips the remaining conditions. If the third condition is checked and if it is true (i.e., if the user clicks on the bottom-left box), then **`color3`** is used. Finally, if the user has pressed the mouse (which we are assuming for this algorithm) but they haven't pressed it in any of the three choice boxes, then `defaultColor` is used.

5.4. Using the `switch` Statement

One disadvantage of the multi-branch `if-else` statement discussed above concerns the efficiency of the manner in which it evaluates its conditions. If the user chooses the upper-left color choice box, the multi-branch `if-else` statement evaluates only one condition, changes the color and moves on. However, if the user chooses the center box, the multi-branch `if-else` statement evaluates two conditions before moving on, and with the bottom box or the default case, three conditions. In general, selecting branch *i* using a multi-branch `if-else` statement requires the evaluation of *i* conditions. Because the evaluation of each condition consumes time, branches that occur later in the multi-branch `if-else` statement take longer to execute than branches that occur earlier in the statement. While this might not matter in simple, non-time-critical applications, it can matter a great deal with time-critical production systems.

To help address this issue, Processing provides the `switch` statement, an alternate multi-branch selection statement that is implemented more efficiently than multi-branch `if-else` statements.

Consider the task of implementing a five-branch selection statement that chooses an appropriate output message for a card guessing game, as specified in the following algorithm.

Given:

- `guess` is set to an integer representing the user's guess.

Algorithm:

```
1. If guess == 2:
   • Print "Got any two's?".
   Otherwise, if guess == 3:
   • Print "Got any three's?".
   Otherwise, if guess == 4:
   • Print "Got any four's?".
   Otherwise, if guess == 5:
   • Print "Got any five's?".
   Otherwise:
   • Print "Invalid guess!".
```

We could implement this algorithm using a multi-branch if statement, but instead we implement it using the `switch` statement as follows.

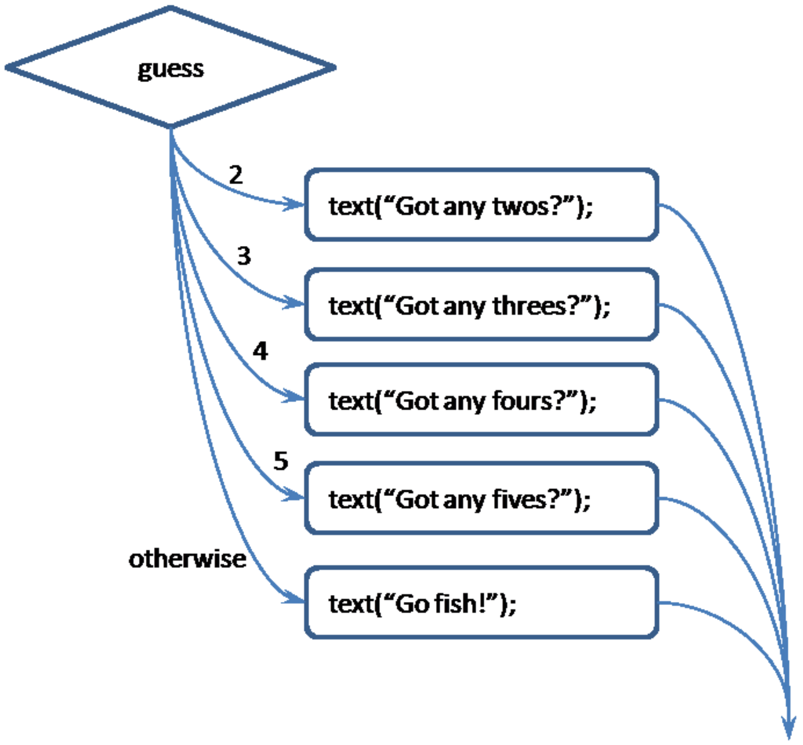
```
switch (guess) {
case 2:
  println("Got any twos?");
  break;
case 3:
  println("Got any threes?");
  break;
case 4:
  println("Got any fours?");
  break;
case 5:
  println("Got any fives?");
  break;
default:
  text("Invalid guess: " + guess);
}
```

This program executes a single `switch` statement that specifies the value of the variable `guess` as its switch expression. It then specifies four cases (for values 2, 3, 4 and 5) and one default case (specified by the `default` keyword). When executed, this `switch` statement checks the value of `guess` and jumps immediately to the case indicated by `guess`'s value. If there is no case specified for `guess`'s value, the `default` case is executed. It is probably easiest to understand its behavior by comparing it to an equivalent multi-branch `if` statement as follows.

Multi-branch `if-else`	**`switch`**
type *variable* = *expression*;	switch (*expression*) {
if (*variable* == *constant*$_1$) {	case *constant1*:
statementList$_1$	*statementList*$_1$
	break;
} else if (*variable* == *constant*$_2$) {	case *constant2*:
statementList$_2$	*statementList*$_2$
	break;
...	...
else if (*variable* == *constant*$_n$) {	case *constant*$_n$:
statementList$_n$	*statementList*$_n$
	break;
} else {	default:
statementList$_{n+1}$	*statementList*$_{n+1}$
}	}

Note that the branches of the `if` statement roughly correspond to the cases of the `switch` statement and that the final `else` clause in the if corresponds to the `default` case.

The `switch` statement provides a couple of advantages over the multi-branch **`if`**: first, its syntax is somewhat cleaner, which makes it easier to understand; second, its execution is more efficient because it only evaluates the switch expression once and then jumps immediately to the correct case. The flow of control through this statement can be visualized as shown in the following flow chart, which assumes that the statement lists in the case clauses end with a `break` statement as programmed above.



As can be seen here, there is only ever one condition that is checked with the switch statement, which means that the efficiency of execution for all the branches is uniform.

There are, however, a few rules to keep in mind when considering the `switch` statement:

1. The `switch` statement requires that its expression return an integer-compatible value. Any expression that returns a value from a finite domain, e.g., integers, booleans, and characters, can be used in a `switch` expression. String, real and reference data types are not integer compatible.[1] In the example given above, the `switch` expression is a single integer variable, `guess`.
2. The branch tests must be equality tests based on the value of the expression and literal values specified in the `case` clauses. In the example given above, the cases are chosen when `guess` is equal to 2, 3, 4, or 5 respectively.
3. `switch` statements implement *drop-through behavior,* which means that when the code segment for the chosen case is executes, control will naturally flow through all the remaining cases in the `switch` block. This is generally not desirable, so the code segments for each case usually end with a statement that forces the control flow to leave the `switch` statement. These statements include the following:
   - The `break` statement, which is used in this example, causes control to go on to the statement following the `switch` statement;
   - The `return` statement causes control to leave the current method – clearly, this statement is only useful in methods;
   - The `throw` statement raises an exception – the control flow for exceptions will be discussed in Chapter 9.

   One useful consequence of this drop-through behavior is that a `switch` statement can specify more than one case clause for a particular statement list. As an example, we could add the following code to the very beginning of the switch statement shown above:

   ```
   switch (guess) {
   case 0: case 1:
     println("Too small");
     break;
   // the other cases follow here unchanged…
   }
   ```

   This code prints "Too small" when `guess` is equal to either 0 or 1. From Processing's point of view, case 0 has no statements, and thus no break statement, so the flow of control moves on to the statement list for case 1.

[1] Java 1.7 supports switching on string values.

`switch` statements use the following general pattern:

`switch` Statement Pattern

```
switch(expression) {
…
case constant_i:
  statementList_i
…
default:
  statementList
}
```

- *expression* is an expression that evaluates to an integer-compatible value;
- Each *constant$_i$* is an integer-compatible constant or literal;
- Each *statementList$_i$* is a sequence of valid statements that can be empty;
- The default case is optional and is executed if none of the other cases holds.