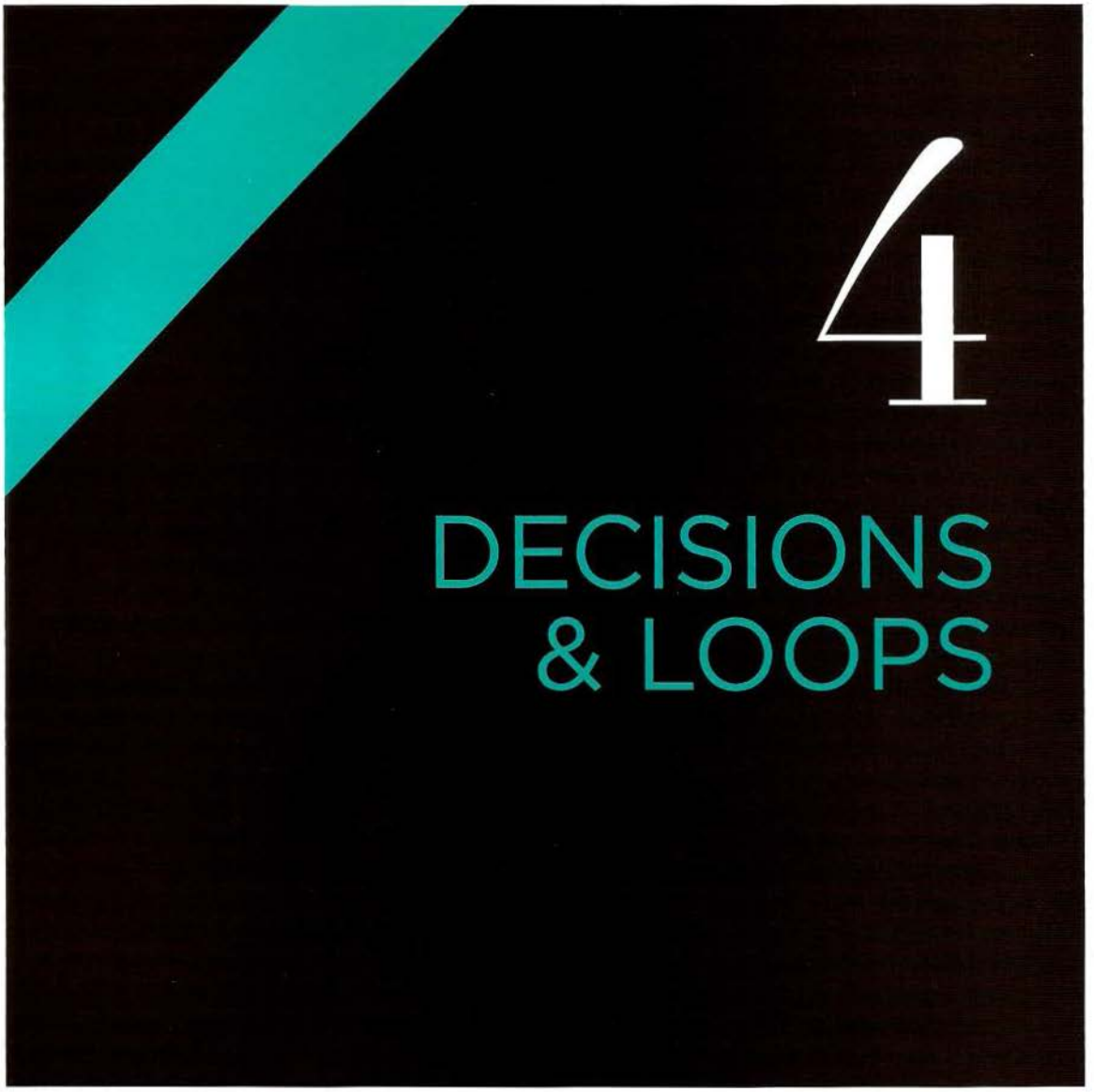

4
DECISIONS
& LOOPS

Looking at a flowchart (for all but the most basic scripts), the code can take more than one path, which means the browser runs different code in different situations. In this chapter, you will learn how to create and control the flow of data in your scripts to handle different situations.

Scripts often need to behave differently depending upon how the user interacts with the web page and/or the browser window itself. To determine which path to take, programmers often rely upon the following three concepts:

EVALUATIONS

You can analyze values in your scripts to determine whether or note they match expected results.

DECISIONS

Using the results of evaluations, you can decide which path your script should go down.

LOOPS

There are also many occasions where you will want to perform the same set of steps repeatedly.

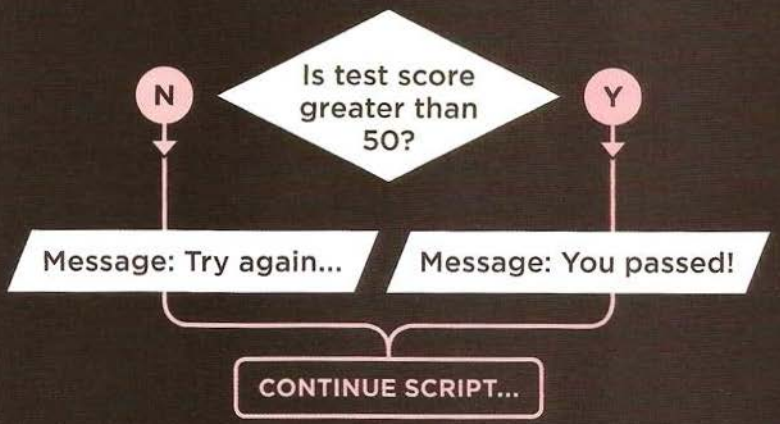

MR PEEK
AT THE ZOO
fischertec
L'HEURE H!
MY LUCKY TRIP
DARTH VADER and son
la mer
Grands Magasins
BULLSEYE!
TARGET PRACTICE FOR YOUR MIND

DECISION MAKING

There are often several places in a script where decisions are made that determine which lines of code should be run next. Flowcharts can help you plan for these occasions.

In a flowchart, the diamond shape represents a point where a decision must be made and the code can take one of two different paths. Each path is made up of a different set of tasks, which means you have to write different code for each situation.

In order to determine which path to take, you set a **condition**. For example, you can check that one value is equal to another, greater than another, or less than another. If the condition returns `true`, you take one path; if it is `false`, you take another path.



In the same way that there are operators to do basic math, or to join two strings, there are **comparison operators** that allow you to compare values and test whether a condition is met or not.

Examples of comparison operators include the greater than (`>`) and less than (`<`) symbols, and double equals sign (`==`) which checks whether two values are the same.

EVALUATING CONDITIONS & CONDITIONAL STATEMENTS

There are two components to a decision:
1: An expression is evaluated, which returns a value
2: A conditional statement says what to do in a given situation

EVALUATION OF A CONDITION

In order to make a decision, your code checks the current status of the script. This is commonly done by comparing two values using a comparison operator which returns a value of **`true`** or **`false`**.

CONDITIONAL STATEMENTS

A conditional statement is based on a concept of if/then/else; *if* a condition is met, *then* your code executes one or more statements, *else* your code does something different (or just skips the step).

CONDITION

```
if (score > 50) {
  document.write('You passed!');
} else {
  document.write('Try again...');
}
```

WHAT THIS IS SAYING:

if the condition returns **true**
execute the statements between the **first** set of curly brackets
otherwise
execute the statements between the **second** set of curly brackets

(You will also meet truthy and falsy values on p167. They are treated as if true or false.)

You can also multiple conditions by combining two or more comparison operators. For example, you can check whether two conditions are both met, or if just one of several conditions is met.

Over the next few pages, you will meet several permutations of the **`if...`** statements, and also a statement called a **switch** statement. Collectively, these are known as **conditional** statements.

COMPARISON OPERATORS: EVALUATING CONDITIONS

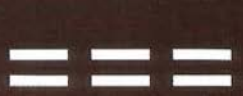
You can evaluate a situation by comparing one value in the script to what you expect it might be. The result will be a Boolean: `true` or `false`.

IS EQUAL TO

This operator compares two values (numbers, strings, or Booleans) to see if they are the same.

`'Hello' == 'Goodbye'` returns `false` because they are *not* the same string.
`'Hello' == 'Hello'` returns `true` because they *are* the same string.

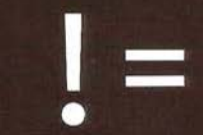
It is usually preferable to use the strict method:



STRICT EQUAL TO

This operator compares two values to check that both the data type and value are the same.

`'3' === 3` returns `false` because they are *not* the same data type or value.
`'3' === '3'` returns `true` because they *are* the same data type and value.



IS NOT EQUAL TO

This operator compares two values (numbers, strings, or Booleans) to see if they are *not* the same.

`'Hello' != 'Goodbye'` returns `true` because they are *not* the same string.
`'Hello' != 'Hello'` returns `false` because they *are* the same string.

It is usually preferable to use the strict method:

STRICT NOT EQUAL TO

This operator compares two values to check that both the data type and value are *not* the same.

`'3' !== 3` returns `true` because they are *not* the same data type or value.
`'3' !== '3'` returns `false` because they *are* the same data type and value.

Programmers refer to the testing or checking of a condition as **evaluating** the condition. Conditions can be much more complex than those shown here, but they usually result in a value of `true` or `false`.

There are a couple of notable exceptions:
i) Every value can be *treated* as true or false even if it is not a Boolean `true` or `false` value (see p167).
ii) In short-circuit evaluation, a condition might not need to run (see p169).

GREATER THAN

This operator checks if the number on the left is *greater than* the number on the right.

`4 > 3` returns `true`
`3 > 4` returns `false`

LESS THAN

This operator checks if the number on the left is *less than* the number on the right.

`4 < 3` returns `false`
`3 < 4` returns `true`

GREATER THAN OR EQUAL TO

This operator checks if the number on the left is *greater than or equal to* the number on the right.

`4 >= 3` returns `true`
`3 >= 4` returns `false`
`3 >= 3` returns `true`

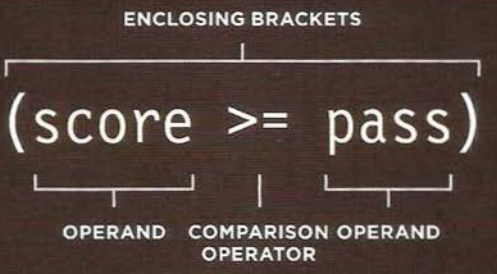
LESS THAN OR EQUAL TO

This operator checks if the number on the left is *less than or equal to* the number on the right.

`4 <= 3` returns `false`
`3 <= 4` returns `true`
`3 <= 3` returns `true`

STRUCTURING COMPARISON OPERATORS

In any condition, there is usually one operator and two operands. The operands are placed on each side of the operator. They can be values or variables. You often see expressions enclosed in brackets.



If you remember back to Chapter 2, this is an example of an **expression** because the condition resolves into a single value: in this case it will be either `true` or `false`.

The enclosing brackets are important when the expression is used as a condition in a comparison operator. But when you are assigning a value to a variable, they are not needed (see right-hand page).

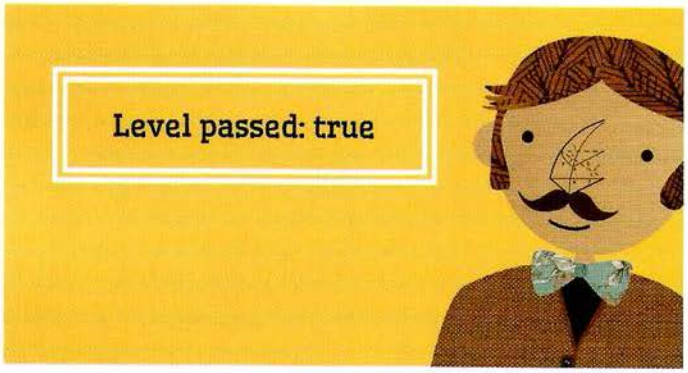
USING COMPARISON OPERATORS

JAVASCRIPT

c04/js/comparison-operator.js

```
var pass = 50;   // Pass mark
var score = 90;  // Score

// Check if the user has passed
var hasPassed = score >= pass;

// Write the message into the page
var el = document.getElementById('answer');
el.textContent = 'Level passed: ' + hasPassed;
```

RESULT



At the most basic level, you can evaluate two variables using a comparison operator to return a `true` or `false` value.

In this example, a user is taking a test, and the script tells the user whether they have passed this round of the test.

The example starts by setting two variables:

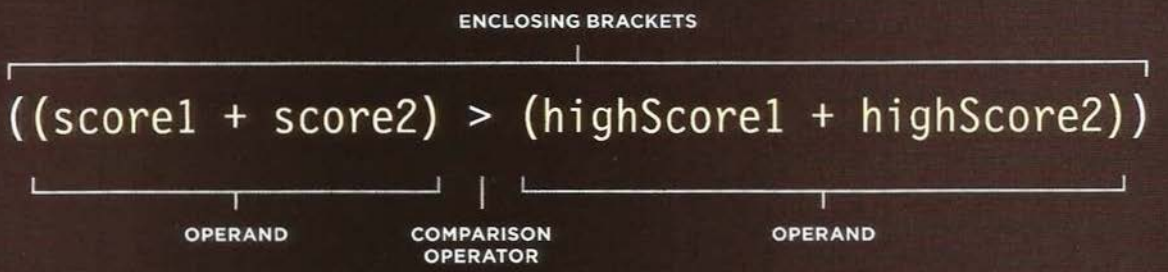
1. `pass` to hold the pass mark
2. `score` to hold the users score

To see if the user has passed, a comparison operator checks whether `score` is greater than or equal to `pass`. The result will be `true` or `false`, and is stored in a variable called `hasPassed`. On the next line, the result is written to the screen.

The last two lines select the element whose `id` attribute has a value of `answer`, and then updates its contents. You will learn more about this technique in the next chapter.

USING EXPRESSIONS WITH COMPARISON OPERATORS

The operand does not have to be a single value or variable name. An operand can be an *expression* (because each expression evaluates into a single value).



COMPARING TWO EXPRESSIONS

In this example, there are two rounds to the test and the code will check if the user has achieved a new high score, beating the previous record.

The script starts by storing the user's scores for each round in variables. Then the highest scores for each round are stored in two more variables.

The comparison operator checks if the user's total score is greater than the highest score for the test and stores the result in a variable called `comparison`.

JAVASCRIPT c04/js/comparison-operator-continued.js

```
var score1 = 90;      // Round 1 score
var score2 = 95;      // Round 2 score
var highScore1 = 75; // Round 1 high score
var highScore2 = 95; // Round 2 high score

// Check if scores are higher than current high scores
var comparison = (score1 + score2) > (highScore1 + highScore2);

// Write the message into the page
var el = document.getElementById('answer');
el.textContent = 'New high score: ' + comparison;
```

RESULT

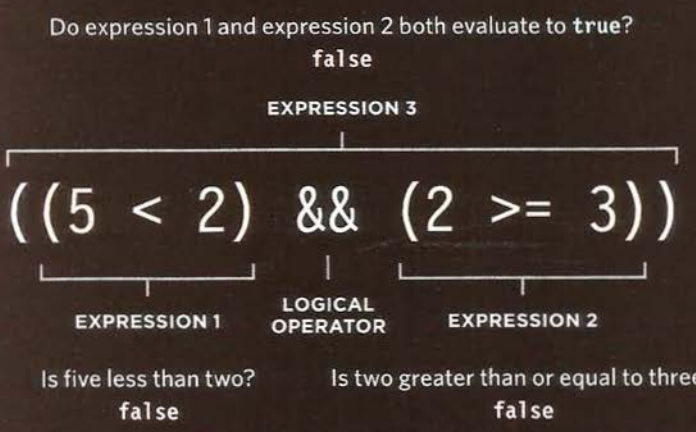
In the comparison operator, the operand on the left calculates the user's total score. The operand on the right adds together the highest scores for each round. The result is then added to the page.

When you assign the result of the comparison to a variable, you do not strictly need the containing parentheses (shown in white on the left-hand page).

Some programmers use them anyway to indicate that the code evaluates into a single value. Others only use containing parentheses when they form part of a condition.

LOGICAL OPERATORS

Comparison operators usually return single values of **true** or **false**. Logical operators allow you to compare the results of more than one comparison operator.



In this one line of code are three expressions, each of which will resolve to the value **true** or **false**.

The expressions on the left and the right both use comparison operators, and both return **false**.

The third expression uses a logical operator (rather than a comparison operator). The logical AND operator checks to see whether both expressions on either side of it return **true** (in this case they do not, so it evaluates to **false**).

&&

LOGICAL AND

This operator tests more than one condition.

`((2 < 5) && (3 >= 2))`
returns **true**

If both expressions evaluate to **true** then the expression returns **true**. If just one of these returns **false**, then the expression will return **false**.

```
true && true  returns true
true && false returns false
false && true  returns false
false && false returns false
```

||

LOGICAL OR

This operator tests at least one condition.

`((2 < 5) || (2 < 1))`
returns **true**

If either expression evaluates to **true**, then the expression returns **true**. If both return **false**, then the expression will return **false**.

```
true || true  returns true
true || false returns true
false || true  returns true
false || false returns false
```

!

LOGICAL NOT

This operator takes a single Boolean value and inverts it.

`!(2 < 1)`
returns **true**

This reverses the state of an expression. If it was **false** (without the ! before it) it would return **true**. If the statement was **true**, it would return **false**.

```
!true  returns false
!false returns true
```

SHORT-CIRCUIT EVALUATION

Logical expressions are evaluated **left** to **right**.
If the first condition can provide enough information to get the answer, then there is no need to evaluate the second condition.

```
false && anything
   ^
```

it has found a **false**

There is no point continuing to determine the other result. They cannot both be **true**.

```
true || anything
   ^
```

it has found a **true**

There is no point continuing because at least one of the values is **true**.

USING LOGICAL AND

In this example, a math test has two rounds. For each round there are two variables: one holds the user's score for that round; the other holds the pass mark for that round.

The logical AND is used to see if the user's score is greater than or equal to the pass mark in both of the rounds of the test. The result is stored in a variable called `passBoth`.

The example finishes off by letting the user know whether or not they have passed both rounds.

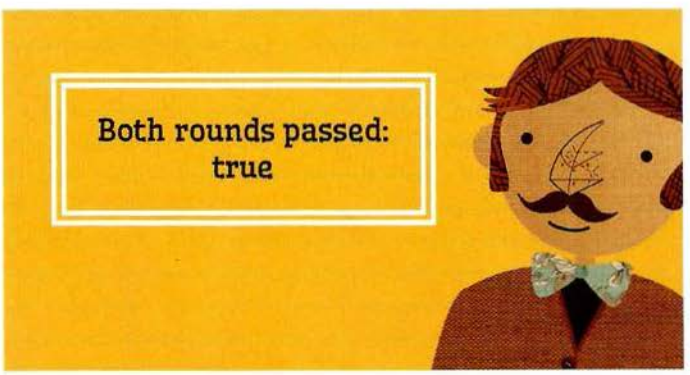
c04/js/logical-and.js

JAVASCRIPT

```
var score1 = 8;    // Round 1 score
var score2 = 8;    // Round 2 score
var pass1 = 6;     // Round 1 pass mark
var pass2 = 6;     // Round 2 pass mark

// Check whether user passed both rounds, store result in variable
var passBoth = (score1 >= pass1) && (score2 >= pass2);

// Create message
var msg = 'Both rounds passed: ' + passBoth;

// Write the message into the page
var el = document.getElementById('answer');
el.textContent = msg;
```

It is rare that you would ever write the Boolean result into the page (like we are doing here). As you will see later in the chapter, it is more likely that you would check a condition, and if it is true, run other statements.

RESULT



USING LOGICAL OR & LOGICAL NOT

Here is the same test but this time using the logical OR operator to find out if the user has passed at least one of the two rounds. If they pass just one round, they do not need to retake the test.

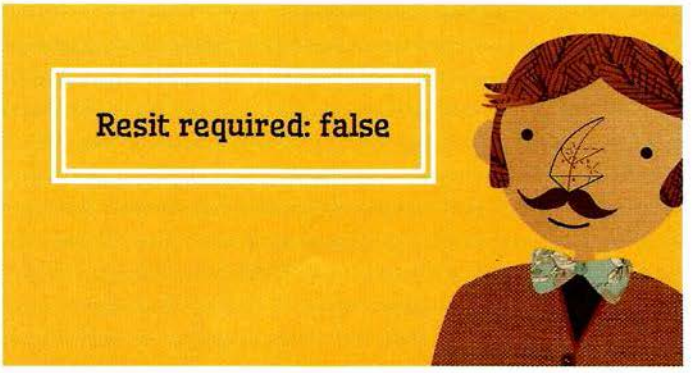
Look at the numbers stored in the four variables at the start of the example. The user has passed both rounds, so the `minPass` variable will hold the Boolean value of `true`.

Next, the message is stored in a variable called `msg`. At the end of the message, the logical NOT will invert the result of the Boolean variable so it is `false`. It is then written into the page.

JAVASCRIPT c04/js/logical-or-logical-not.js

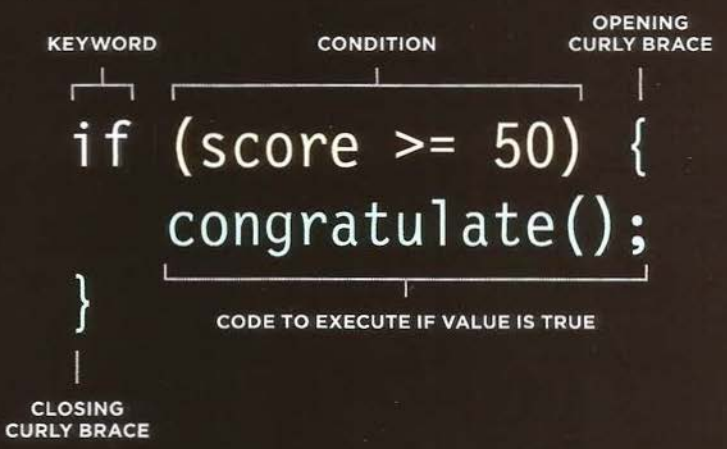
```
var score1 = 8;    // Round 1 score
var score2 = 8;    // Round 2 score
var pass1 = 6;     // Round 1 pass mark
var pass2 = 6;     // Round 2 pass mark

// Check whether user passed one of the two rounds, store result in variable
var minPass = ((score1 >= pass1) || (score2 >= pass2));

// Create message
var msg = 'Resit required: ' + !(minPass);

// Write the message into the page
var el = document.getElementById('answer');
el.textContent = msg;
```

RESULT



IF STATEMENTS

The **if** statement evaluates (or checks) a condition. If the condition evaluates to **true**, any statements in the subsequent code block are executed.



If the condition evaluates to **true**, the following code block (the code in the next set of curly braces) is executed.

If the condition resolves to **false**, the statements in that code block are *not* run. (The script continues to run from the end of the next code block.)

USING IF STATEMENTS

JAVASCRIPT c04/js/if-statement.js

```
var score = 75;      // Score
var msg;             // Message

if (score >= 50) {  // If score is 50 or higher
  msg = 'Congratulations!';
  msg += ' Proceed to the next round.';
}
var el = document.getElementById('answer');
el.textContent = msg;
```

RESULT

In this example, the `if` statement is checking if the value currently held in a variable called `score` is 50 or more.

In this case, the statement evaluates to `true` (because the score is 75, which is greater than 50). Therefore, the contents of the statements within the subsequent code block are run, creating a message that congratulates the user and tells them to proceed.

After the code block, the message is written to the page.

If the value of the `score` variable had been less than 50, the statements in the code block would not have run, and the code would have continued on to the next line after the code block.

JAVASCRIPT c04/js/if-statement-with-function.js

```
var score = 75;     // Score
var msg = '';       // Message

function congratulate() {
    msg += 'Congratulations! ';
}

if (score >= 50) {  // If score is 50 or more
  congratulate();
  msg += 'Proceed to the next round.';
}
var el = document.getElementById('answer');
el.innerHTML = msg;
```

On the left is an alternative version of the same example that demonstrates how lines of code do not always run in the order you expect them to. ***If*** the condition is met then:

1. The first statement in the code block calls the `congratulate()` function.

2. The code within the `congratulate()` function runs.

3. The second line within the `if` statement's code block runs.

IF...ELSE STATEMENTS

The **`if...else`** statement checks a condition.
If it resolves to **`true`** the first code block is executed.
If the condition resolves to **`false`** the second code block is run instead.

```
if (score >= 50) {
    congratulate();
}
else {
    encourage();
}
```

CODE TO EXECUTE IF VALUE IS TRUE

CODE TO EXECUTE IF VALUE IS FALSE

- CONDITIONAL STATEMENT
- CONDITION
- IF CODE BLOCK
- ELSE CODE BLOCK

USING IF...ELSE STATEMENTS

JAVASCRIPT c04/js/if-else-statement.js

```
var pass = 50;        // Pass mark
var score = 75;       // Current score
var msg;              // Message

// Select message to write based on score
if (score >= pass) {
  msg = 'Congratulations, you passed!';
} else {
  msg = 'Have another go!';
}

var el = document.getElementById('answer');
el.textContent = msg;
```

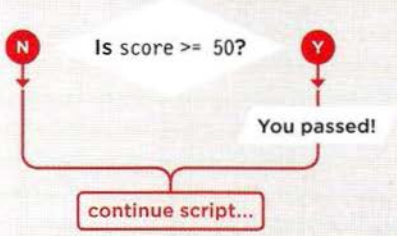
RESULT

Here you can see that an `if...else` statement allows you to provide two sets of code:
1. one set if the condition evaluates to `true`
2. another set if the condition is `false`

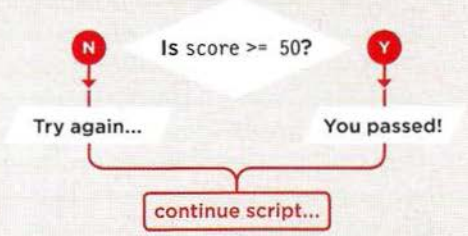
In this test, there are two possible outcomes: a user can either get a score equal to or greater than the pass mark (which means they pass), or they can score less than the pass mark (which means they fail). One response is required for each eventuality. The response is then written to the page.

Note that the statements inside an `if` statement should be followed by a semicolon, but there is no need to place one after the closing curly brace of the code blocks.

An `if` statement only runs a set of statements if the condition is `true`:



An `if...else` statement runs one set of code if the condition is `true` or a different set if it is `false`:



SWITCH STATEMENTS

A **switch** statement starts with a variable called the **switch value**. Each case indicates a possible value for this variable and the code that should run if the variable matches that value.

```
switch (level) {

  case 'One':
    title = 'Level 1';
    break;

  case 'Two':
    title = 'Level 2';
    break;

  case 'Three':
    title = 'Level 3';
    break;

  default:
    title = 'Test';
    break;

}
```

Here, the variable named **level** is the switch value. If the value of the **level** variable is the string **One**, then the code for the first case is executed. If it is **Two**, the second case is executed. If it is **Three**, the third case is executed. If it is none of these, the code for the **default** case is executed.

The entire statement lives in one code block (set of curly braces), and a colon separates the option from the statements that are to be run if the case matches the switch value.

At the end of each case is the **break** keyword. It tells the JavaScript interpreter that it has finished with this **switch** statement and to proceed to run any subsequent code that appears after it.

IF... ELSE

- There is no need to provide an **else** option. (You can just use an **if** statement.)
- With a series of **if** statements, they are all checked *even if* a match has been found (so it performs more slowly than **switch**).

VS.

SWITCH

- You have a **default** option that is run if none of the cases match.
- If a match is found, that code is run; then the **break** statement stops the rest of the **switch** statement running (providing better performance than multiple **if** statements).

USING SWITCH STATEMENTS

JAVASCRIPT c04/js/switch-statement.js

```
var msg;         // Message
var level = 2;   // Level

// Determine message based on level
switch (level) {
case 1:
    msg = 'Good luck on the first test';
    break;

case 2:
    msg = 'Second of three - keep going!';
    break;

case 3:
    msg = 'Final round, almost there!';
    break;

default:
    msg = 'Good luck!';
    break;
}

var el = document.getElementById('answer');
el.textContent = msg;
```

RESULT

In this example, the purpose of the `switch` statement is to present the user with a different message depending on which level they are at. The message is stored in a variable called `msg`.

The variable called `level` contains a number indicating which level the user is on. This is then used as the switch value. (The switch value could also be an expression.)

In the following code block (inside the curly braces), there are three options for what the value of the `level` variable might be: the numbers 1, 2, or 3.

If the value of the `level` variable is the number 1, the value of the `msg` variable is set to `'Good luck on the first test'`.

If the value is 2, it will read: `'Second of three - keep going!'`

If the value is 3, the message will read: `'Final round, almost there!'`

If no match is found, then the value of the `msg` variable is set to `'Good luck!'`

Each case ends with the `break` keyword which will tell the JavaScript interpreter to skip the rest of this code block and continue onto the next.

TYPE COERCION & WEAK TYPING

If you use a data type JavaScript did not expect, it tries to make sense of the operation rather than report an error.

JavaScript can convert data types behind the scenes to complete an operation. This is known as **type coercion**. For example, a string `'1'` could be converted to a number 1 in the following expression: (`'1' > 0`). As a result, the above expression would evaluate to `true`.

JavaScript is said to use **weak typing** because the data type for a value can change. Some other languages require that you specify what data type each variable will be. They are said to use **strong typing**.

Type coercion can lead to unexpected values in your code (and also cause errors). Therefore, when checking if two values are equal, it is considered better to use strict equals operators `===` and `!==` rather than `==` and `!=` as these strict operators check that the value and data types match.

DATA TYPE	PURPOSE
`string`	Text
`number`	Number
`Boolean`	`true` or `false`
`null`	Empty value
`undefined`	Variable has been declared but not yet assigned a value

`NaN` is a value that is counted as a number. You may see it when a number is expected, but is not returned, e.g., (`'ten'/2`) results in `NaN`.

TRUTHY & FALSY VALUES

Due to type coercion, every value in JavaScript can be treated as if it were true or false; and this has some interesting side effects.

FALSY VALUES

VALUE	DESCRIPTION
`var highScore = false;`	The traditional Boolean `false`
`var highScore = 0;`	The number zero
`var highScore = '';`	`NaN` (Not a Number)
`var highScore = 10/'score';`	Empty value
`var highScore;`	A variable with no value assigned to it

Almost everything else evaluates to truthy...

TRUTHY VALUES

VALUE	DESCRIPTION
`var highScore = true;`	The traditional Boolean `true`
`var highScore = 1;`	Numbers other than zero
`var highScore = 'carrot';`	Strings with content
`var highScore = 10/5;`	Number calculations
`var highScore = 'true';`	`true` written as a string
`var highScore = '0';`	Zero written as a string
`var highScore = 'false';`	`false` written as a string

Falsy values are treated *as if* they are `false`. The table to the left shows a `highScore` variable with a series of values, all of which are falsy.

Falsy values can also be treated as the number 0.

Truthy values are treated *as if* they are `true`. Almost everything that is not in the falsy table can be treated as if it were true.

Truthy values can also be treated as the number 1.

In addition, the presence of an object or an array is usually considered truthy, too. This is commonly used when checking for the presence of an element in a page.

The next page will explain more about why these concepts are important.

CHECKING EQUALITY & EXISTENCE

Because the presence of an object or array can be considered truthy, it is often used to check for the existence of an element within a page.

A **unary operator** returns a result with just one operand. Here you can see an `if` statement checking for the presence of an element. If the element is found, the result is truthy, so the first set of code is run. If it is not found, the second set is run instead.

```
if (document.getElementById('header')) {
  // Found: do something
} else {
  // Not found: do something else
}
```

Those new to JavaScript often think the following would do the same:
`if (document.getElementById('header') == true)`
but `document.getElementById('header')` would return an object which is a truthy value but it is *not* equal to a Boolean value of `true`.

Because of type coercion, the strict equality operators === and !== result in fewer unexpected values than == and != do.

If you use == the following values can be considered equal: `false`, `0`, and `''` (empty string). However, they are not equivalent when using the strict operators.

EXPRESSION	RESULT
`(false == 0)`	`true`
`(false === 0)`	`false`
`(false == '')`	`true`
`(false === '')`	`false`
`(0 == '')`	`true`
`(0 === '')`	`false`

Although `null` and `undefined` are both falsy, they are not equal to anything other than themselves. Again, they are not equivalent when using strict operators.

EXPRESSION	RESULT
`(undefined == null)`	`true`
`(null == false)`	`false`
`(undefined == false)`	`false`
`(null == 0)`	`false`
`(undefined == 0)`	`false`
`(undefined === null)`	`false`

Although `NaN` is considered falsy, it is not equivalent to anything; it is not even equivalent to itself (since `NaN` is an undefinable number, two cannot be equal).

EXPRESSION	RESULT
`(Nan == null)`	`false`
`(NaN == NaN)`	`false`

SHORT CIRCUIT VALUES

Logical operators are processed left to right. They short-circuit (stop) as soon as they have a result – but they return the value that stopped the processing (not necessarily `true` or `false`).

On line 1, the variable `artist` is given a value of `Rembrandt`.
On line 2, if the variable `artist` has a value, then `artistA` will be given the same value as `artist` (because a non-empty string is truthy).

```
var artist = 'Rembrandt';
var artistA = (artist || 'Unknown');
```

If the string is empty (see below), `artistA` becomes a string `'Unknown'`.

```
var artist = '';
var artistA = (artist || 'Unknown');
```

You could even create an empty object if artist does not have a value:

```
var artist = '';
var artistA = (artist || {});
```

Logical operators will not always return `true` or `false`, because:

- They return the value that stopped processing.
- That value might have been treated as truthy or falsy although it was not a Boolean.

Programmers use this creatively (for example, to set values for variables or even create objects).

Here are three values. If any one of them is considered truthy, the code inside the `if` statement will execute. When the script encounters `valueB` in the logical operator, it will short circuit because the number 1 is considered truthy and the subsequent code block is executed.

```
valueA = 0;
valueB = 1;
valueC = 2;

if (valueA || valueB || valueC) {
  // Do something here
}
```

This technique could also be used to check for the existence of elements within a page, as shown on p168.

As soon as a truthy value is found, the remaining options are not checked. Therefore, experienced programmers often:

- Put the code most likely to return true *first* in OR operations, and false answers first in AND operations.
- Place the options requiring the most processing power last, just in case another value returns `true` and they do not need to be run.

LOOPS

Loops check a condition. If it returns **true**, a code block will run. Then the condition will be checked again and if it still returns **true**, the code block will run again. It repeats until the condition returns **false**. There are three common types of loops:

FOR

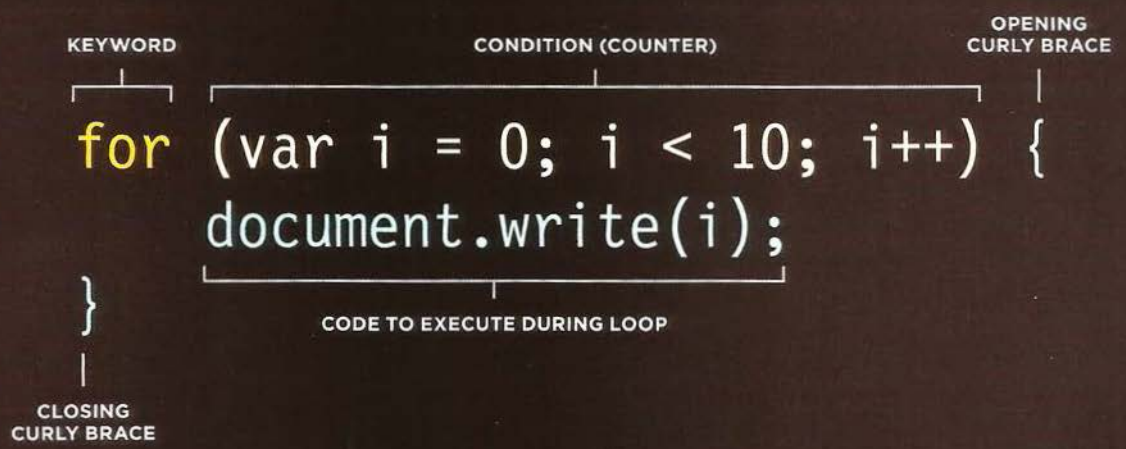
If you need to run code a specific number of times, use a **for** loop. (It is the most common loop.) In a **for** loop, the condition is usually a counter which is used to tell how many times the loop should run.

WHILE

If you do not know how many times the code should run, you can use a **while** loop. Here the condition can be something other than a counter, and the code will continue to loop for as long as the condition is **true**.

DO WHILE

The **do...while** loop is very similar to the **while** loop, but has one key difference: it will always run the statements inside the curly braces at least once, even if the condition evaluates to **false**.



This is a **for** loop. The condition is a counter that counts to ten. The result would write "0123456789" to the page.

If the variable **i** is less than ten, the code inside the curly braces is executed. Then the counter is incremented.

The condition is checked again, if **i** is less than ten it runs again. The next three pages show how this loop works in greater detail.

LOOP COUNTERS

A **for** loop uses a counter as a condition.
This instructs the code to run a specified number of times.
Here you can see the condition is made up of three statements:

INITIALIZATION

Create a variable and set it to **0**. This variable is commonly called **i**, and it acts as the counter.

```
var i = 0;
```

The variable is only created the first time the loop is run. (You may also see the variable called **index**, rather than just **i**.)

You will sometimes see this variable declared before the condition. The following is the same and it is mainly a preference of the coder.

```
var i;
for (i = 0; i < 10; i++) {
  // Code goes here
}
```

CONDITION

The loop should continue to run until the counter reaches a specified number.

```
i < 10;
```

The value of **i** was initially set to **0**, so in this case the loop will run 10 times before stopping.

The condition may also use a variable that holds a number. If a variable called **rounds** held the number of rounds in a test and the loop ran once for each round, the condition would be:

```
var rounds = 3;
i < (rounds);
```

UPDATE

Every time the loop has run the statements in the curly braces, it adds one to the counter.

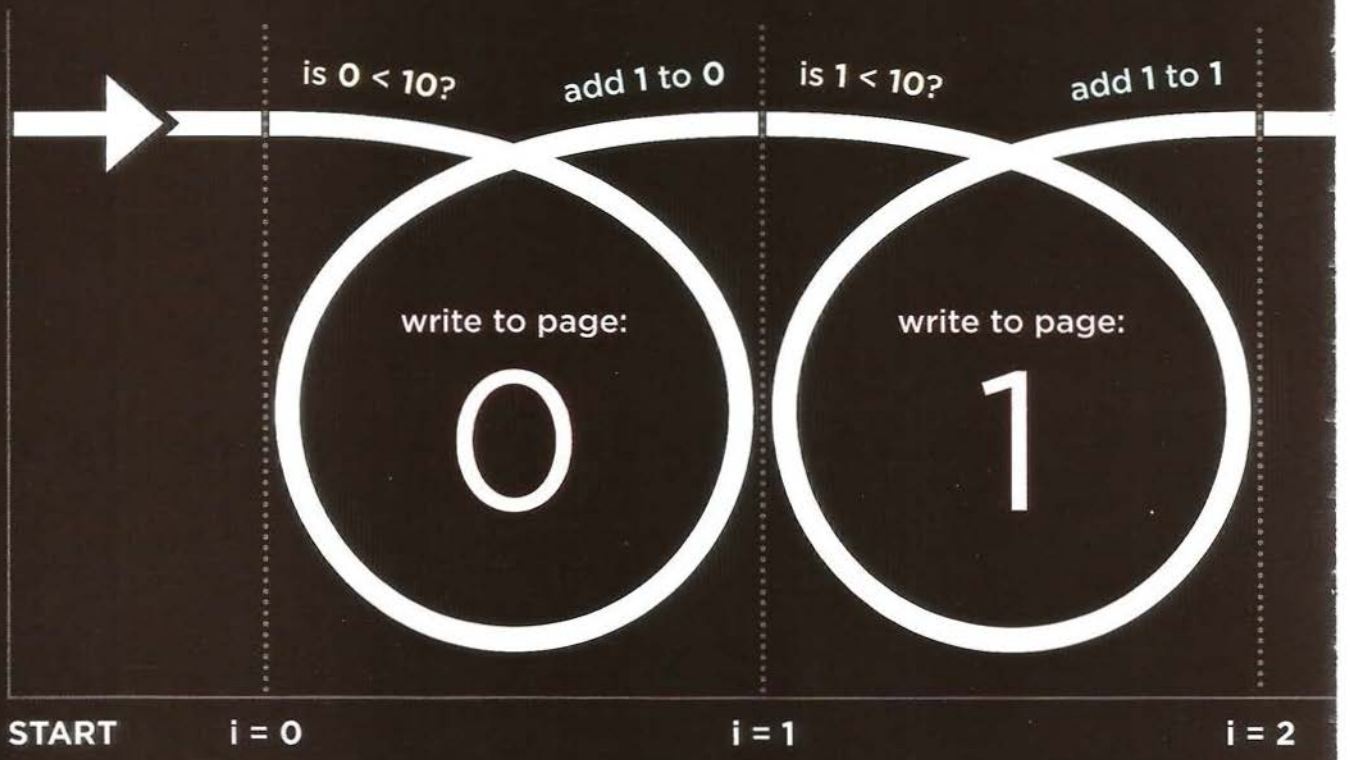
```
i++
```

One is added to the counter using the increment (**++**) operator.

Another way of reading this is that it says, "Take the variable **i**, and add one using the **++** operator."

It is also possible for loops to count downwards using the decrement operator (**--**).

LOOPING

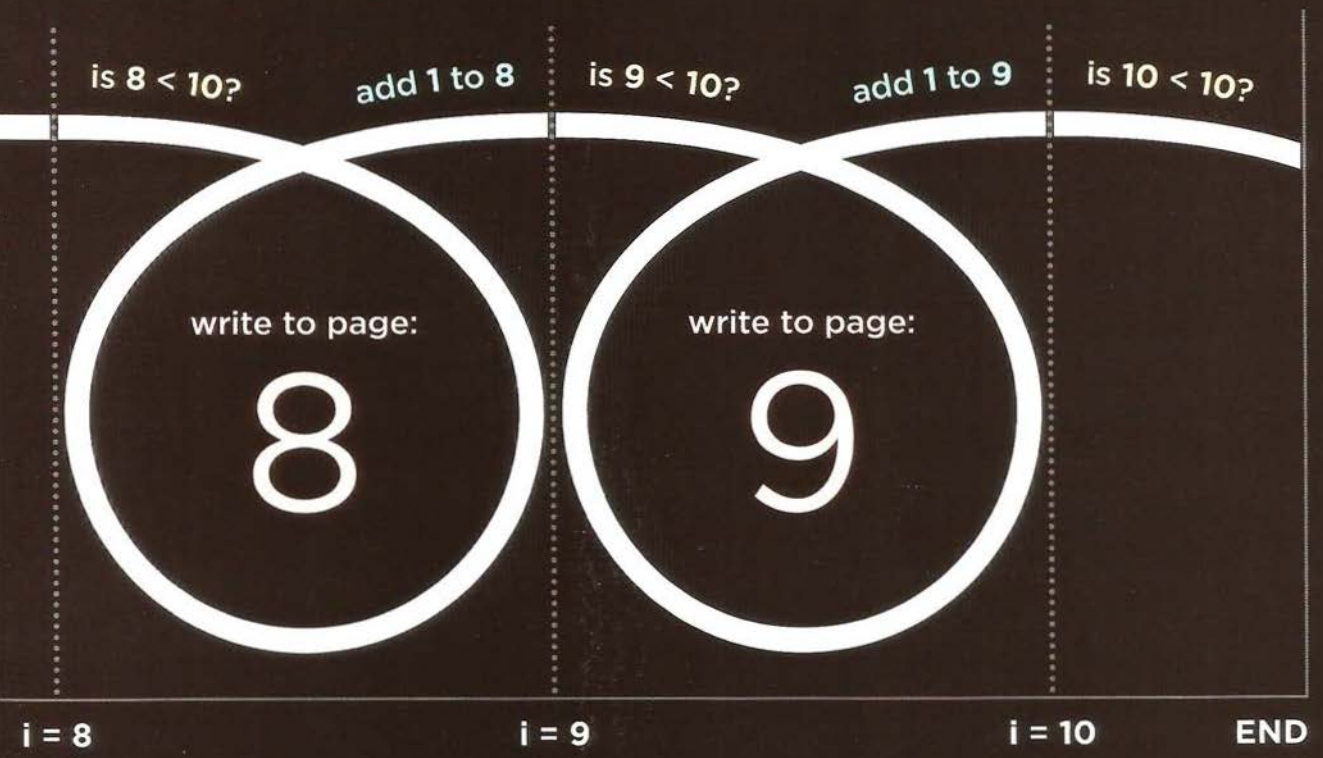


The first time the loop is run, the variable `i` (the counter) is assigned a value of zero.

Every time the loop is run, the condition is checked. Is the variable `i` less than 10?

Then the code inside the loop (the statements between the curly brackets) is run.

```
for (var i = 0; i < 10; i++) {
    document.write(i);
}
```



The variable `i` can be used inside the loop. Here it is used to write a number to the page.

When the statements have finished, the variable `i` is incremented by 1.

When the condition is no longer true, the loop ends. The script moves to the next line of code.

KEY LOOP CONCEPTS

Here are three points to consider when you are working with loops. Each is illustrated in examples on the following three pages.

KEYWORDS

You will commonly see these two keywords used with loops:

`break`

This keyword causes the termination of the loop and tells the interpreter to go onto the next statement of code outside of the loop. (You may also see it used in functions.)

`continue`

This keyword tells the interpreter to continue with the current iteration, and then check the condition again. (If it is `true`, the code runs again.)

LOOPS & ARRAYS

Loops are very helpful when dealing with arrays if you want to run the same code for each item in the array.

For example, you might want to write the value of each item stored in an array into the page.

You may not know how many items will be in an array when writing a script, but, when the code runs, it can check the total number of items in a loop. That figure can then be used in the counter to control how many times a set of statements is run.

Once the loop has run the right number of times, the loop stops.

PERFORMANCE ISSUES

It is important to remember that when a browser comes across JavaScript, it will stop doing anything else until it has processed that script.

If your loop is dealing with only a small number of items, this will not be an issue. If, however, your loop contains a lot of items, it can make the page slower to load.

If the condition never returns `false`, you get what is commonly referred to as an **infinite loop**. The code will not stop running until your browser runs out of memory (breaking your script).

Any variable you can define outside of the loop and that does not change *within* the loop should be defined outside of it. If it were declared inside the loop, it would be recalculated every time the loop ran, needlessly using resources.

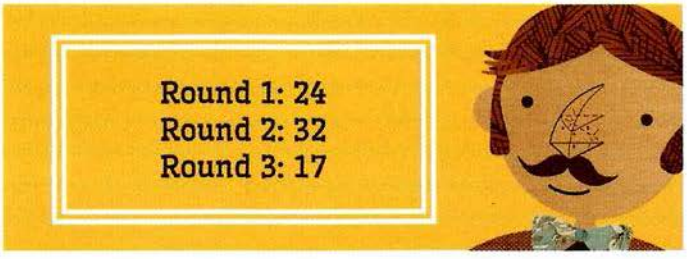
USING FOR LOOPS

JAVASCRIPT

c04/js/for-loop.js

```
var scores = [24, 32, 17];       // Array of scores
var arrayLength = scores.length;// Items in array
var roundNumber = 0;             // Current round
var msg = '';                    // Message
var i;                           // Counter

// Loop through the items in the array
for (i = 0; i < arrayLength; i++) {

  // Arrays are zero based (so 0 is round 1)
  // Add 1 to the current round
  roundNumber = (i + 1);

  // Write the current round to message
  msg += 'Round ' + roundNumber + ': ';

  // Get the score from the scores array
  msg += scores[i] + '<br />';
}

document.getElementById('answer').innerHTML = msg;
```

RESULT



The counter and array both start from 0 (rather than 1). So, within the loop, to select the current item from the array, you use the counter variable `i` to specify the item from the array, e.g., `scores[i]`. But remember that it is a number lower then you might expect (e.g., first iteration is 0, second is 1).

A `for` loop is often used to loop through the items in an array.

In this example, the scores for each round of a test are stored in an array called `scores`.

The total number of items in the array is stored in a variable called `arrayLength`. This number is obtained using the `length` property of the array.

There are three more variables: `roundNumber` holds the round of the test; `msg` holds the message to display; `i` is the counter (declared outside the loop).

The loop starts with the `for` keyword, then contains the condition inside the parentheses. As long as the counter is less than the total number of items in the array, the contents of the curly braces will continue to run. Each time the loop runs, the round number is increased by 1.

Inside the curly braces are rules that write the round number and the score to the `msg` variable. The variables declared outside of the loop are used within the loop.

The `msg` variable is then written into the page. It contains HTML so the `innerHTML` property is used to do this. Remember, p228 will talk about security issues relating to this property.

USING WHILE LOOPS

Here is an example of a `while` loop. It writes out the 5 times table. Each time the loop is run, another calculation is written into the variable called `msg`.

This loop will continue to run for as long as the condition in the parentheses is true. That condition is a counter indicating that, as long as the variable `i` remains less than 10, the statements in the subsequent code block should run.

Inside the code block there are two statements:

The first statement uses the `+=` operator, which is used to add new content to the `msg` variable. Each time the loop runs, a new calculation and line break is added to the end of the message being stored in it. So `+=` works as a shorthand for writing:
`msg = msg + 'new msg'`
(See bottom of the next page for a breakdown of this statement.)

The second statement increments the counter variable by one. (This is done inside the loop rather than with the condition.)

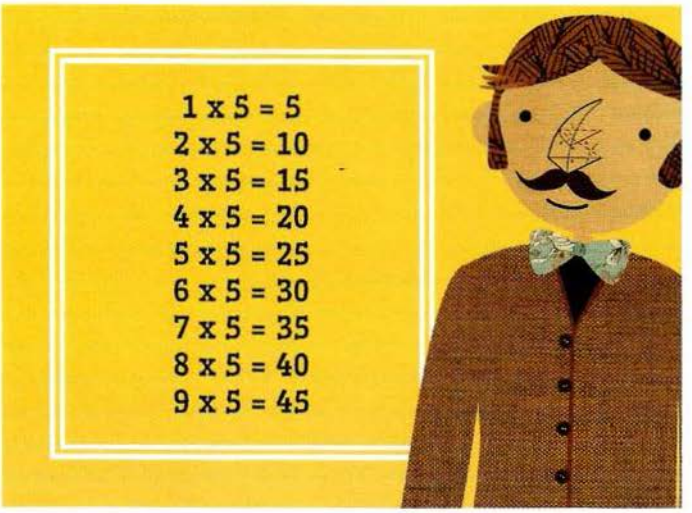
When the loop has finished, the interpreter goes to the next line of code, which writes the `msg` variable to the page.

c04/js/while-loop.js

JAVASCRIPT

```
var i = 1;          // Set counter to 1
var msg = '';       // Message

// Store 5 times table in a variable
while (i < 10) {
  msg += i + ' x 5 = ' + (i * 5) + '<br />';
  i++;
}

document.getElementById('answer').innerHTML = msg;
```

RESULT



In this example, the condition specifies that the code should run nine times. A more typical use of a `while` loop would be when you *do not know* how many times you want the code to run. It should continue to run as long as a condition is met.

USING DO WHILE LOOPS

JAVASCRIPT c04/js/do-while-loop.js

```
var i = 1;        // Set counter to 1
var msg = '';     // Message

// Store 5 times table in a variable
do {
  msg += i + ' x 5 = ' + (i * 5) + '<br />';s
  i++;
} while (i < 1);
// Note how this is already 1 and it still runs

document.getElementById('answer').innerHTML = msg;
```

RESULT

Breaking down the first statement in these examples:

```
msg += i + ' x 5 = ' + (i * 5) + '<br />';
```

1 2 3 4 5 6

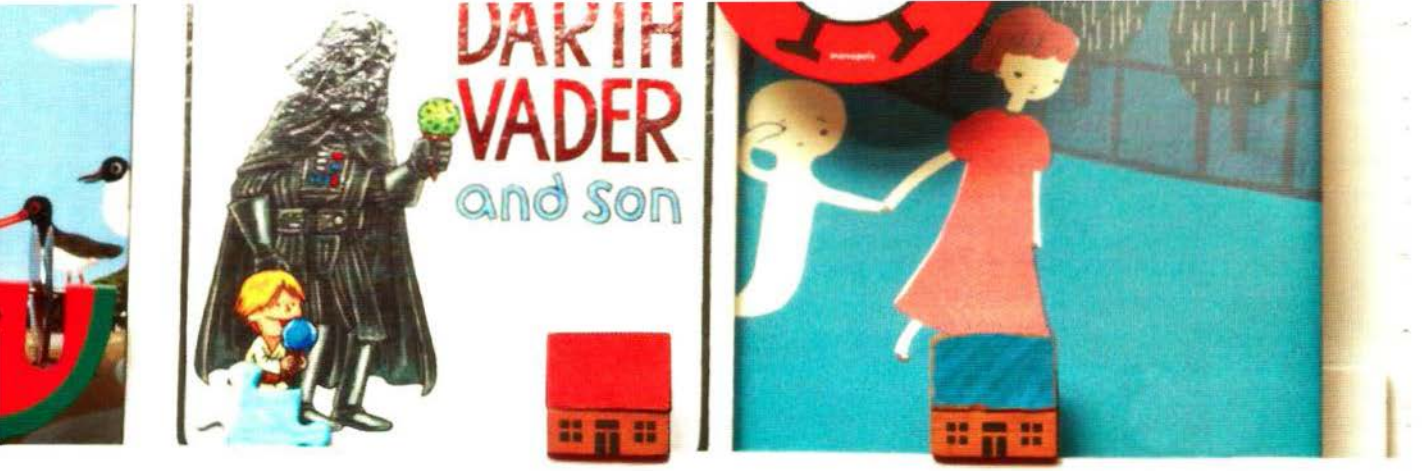
1. Take variable called `msg`
2. Add to the following to its value
3. The number in the counter
4. Write out the string **x 5 =**
5. The counter multiplied by **5**
6. Add a line break

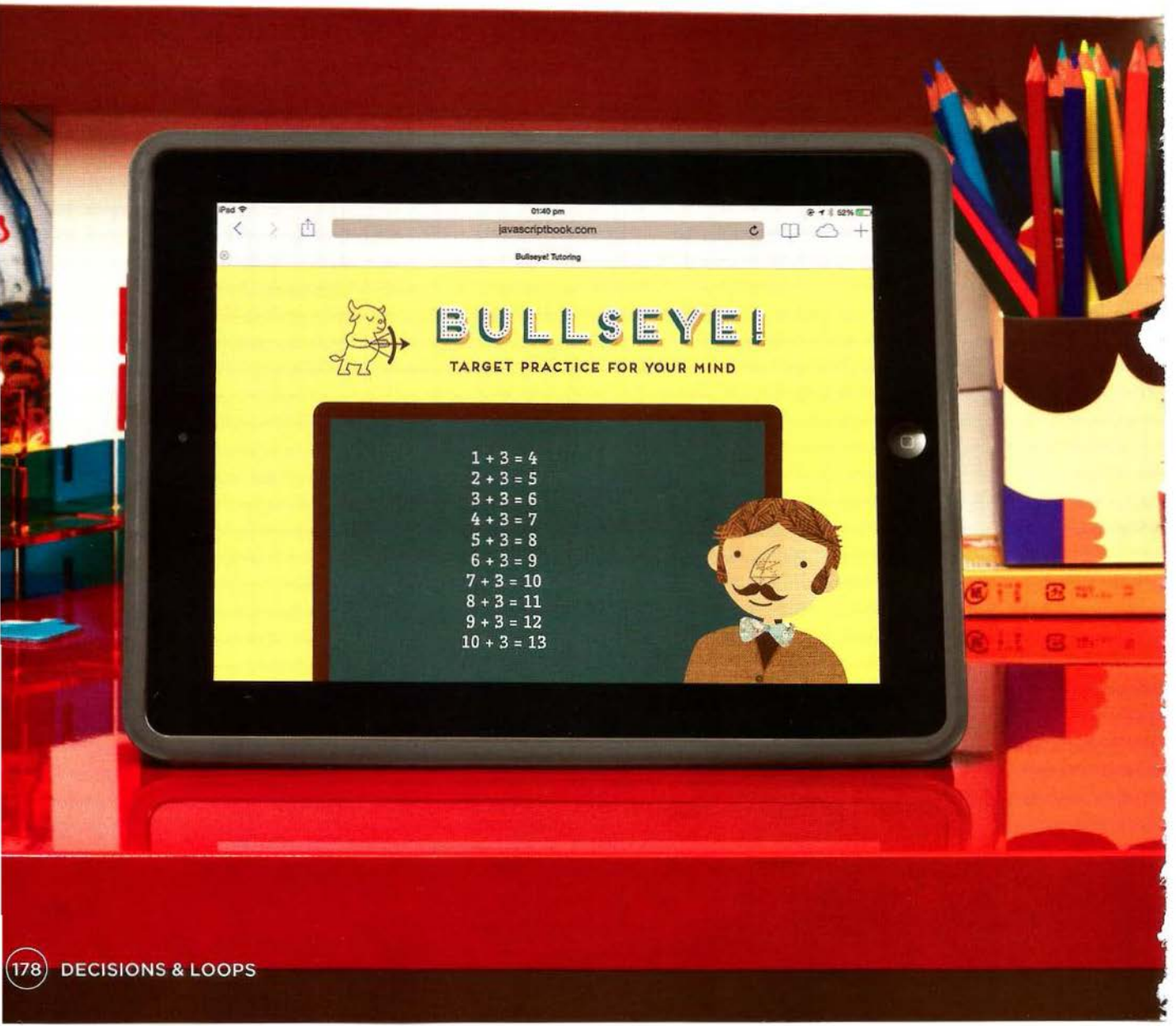
The key difference between a `while` loop and a `do while` loop is that the statements in the code block come *before* the condition. This means that those statements are run once whether or not the condition is met.

If you take a look at the condition, it is checking that the value of the variable called `i` is less than 1, but that variable has already been set to a value of 1.

Therefore, in this example the result is that the 5 times table is written out once, even though the counter is not less than 1.

Some people like to write `while` on a separate line from the closing curly brace before it.


DARTH
VADER
and son


iPad
01:40 pm
52%
javascriptbook.com
Bullseye! Tutoring
BULLSEYE!
TARGET PRACTICE FOR YOUR MIND
1 + 3 = 4
2 + 3 = 5
3 + 3 = 6
4 + 3 = 7
5 + 3 = 8
6 + 3 = 9
7 + 3 = 10
8 + 3 = 11
9 + 3 = 12
10 + 3 = 13

EXAMPLE
DECISIONS & LOOPS

In this example, the user can either be shown addition or multiplication of a given number. The script demonstrates the use of both conditional logic and loops.

The example starts with two variables:

1. `number` holds the number that the calculations will be performed with (in this case it is the number 3)
2. `operator` indicates whether it should be addition or multiplication (in this case it is performing addition)

An `if...else` statement is used to decide whether to perform addition or multiplication with the number. If the variable called `operator` has the value `addition`, the numbers will be added together; otherwise they will be multiplied.

Inside the conditional statement, a `while` loop is used to calculate the results. It will run 10 times because the condition is checking whether the value of the counter is less than 11.

EXAMPLE
DECISIONS & LOOPS

c04/example.html

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bullseye! Tutoring</title>
    <link rel="stylesheet" href="css/c04.css" />
  </head>
  <body>
    <section id="page2">
      <h1>Bullseye</h1>
      <img src="images/teacher.png" id="teacher2" alt="" />
      <section id="blackboard"></section>
    </section>
    <script src="js/example.js"></script>
  </body>
</html>
```

The HTML for this example is very slightly different than the other examples in this chapter because there is a blackboard which the table is written onto.

You can see the script is added to the page just before the closing </body> tag.

EXAMPLE
DECISIONS & LOOPS

JAVASCRIPT

c04/js/example.js

```
var table = 3;                          // Unit of table
var operator = 'addition';              // Type of calculation (defaults to addition)
var i = 1;                              // Set counter to 1
var msg = '';                           // Message

if (operator === 'addition') {          // If the operator variable says addition
  while (i < 11) {                      // While counter is less than 11
    msg += i + ' + ' + table + ' = ' + (i + table) + '<br />'; // Calculation
    i++;                                // Add 1 to the counter
  }
} else {                                // Otherwise
  while (i < 11) {                      // While counter is less than 11
    msg += i + ' x ' + table + ' = ' + (i * table) + '<br />'; // Calculation
    i++;                                // Add 1 to the counter
  }
}

// Write the message into the page
var el = document.getElementById('blackboard');
el.innerHTML = msg;
```

If you read the comments in the code, you can see how this example works. The script starts by declaring four variables and setting values for them.

Then, an `if` statement checks whether the value of the variable called `operator` is `addition`. If it is, it uses a `while` loop to perform the calculations and store the results in a variable called `msg`.

If you change the value of the `operator` variable to anything other than `addition`, the conditional statement will select the second set of statements. These also contain a `while` loop, but this time it will perform multiplication (rather than addition).

When one of the loops has finished running, the last two lines of the script select the element whose `id` attribute has a value of `blackboard`, and updates the the page with the content of the `msg` variable.

SUMMARY

DECISIONS & LOOPS

- Conditional statements allow your code to make decisions about what to do next.
- Comparison operators (`===`, `!==`, `==`, `!=`, `<`, `>`, `<=`, `=>`) are used to compare two operands.
- Logical operators allow you to combine more than one set of comparison operators.
- `if...else` statements allow you to run one set of code if a condition is true, and another if it is false.
- `switch` statements allow you to compare a value against possible outcomes (and also provides a default option if none match).
- Data types can be coerced from one type to another.
- All values evaluate to either truthy or falsy.
- There are three types of loop: `for`, `while`, and `do...while`. Each repeats a set of statements.