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.

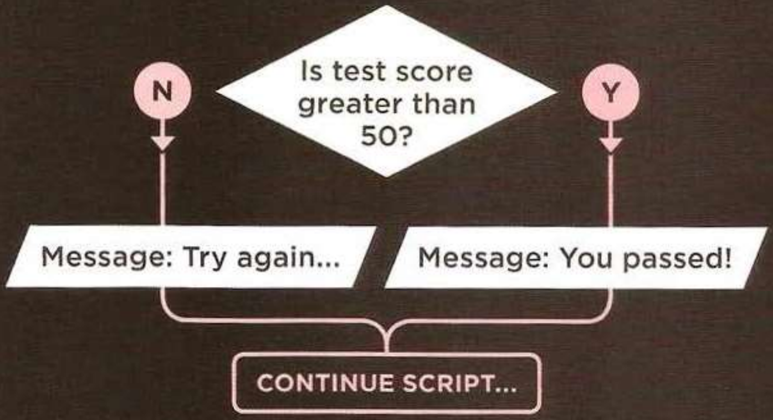
MR PEEK AT THE ZOO
fischertec
L'HEURE H!
mon album animé la mer
DARTH VADER and Son
MY LUCKY TRIP
Grands Magasins
BULLSEYE!
TARGET PRACTICE FOR YOUR MIND
Gaspard & Lisa

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:

!=

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 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.

!==

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`

<=

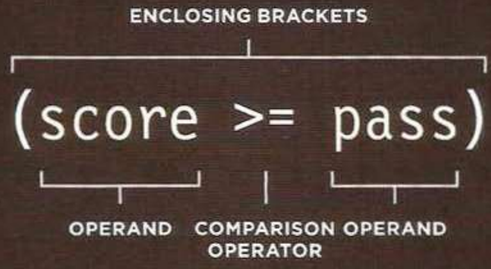
LESS THAN OR EQUAL TO

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`.

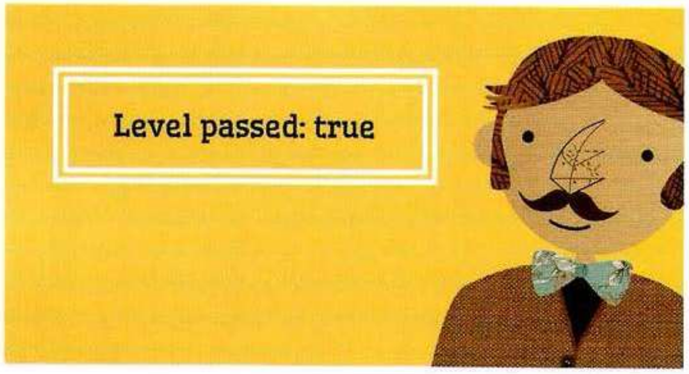
The enclosing brackets are important when the expression is used as a condition in a comparison operator. But when you are assigning a value to a variable, they are not needed (see right-hand page).

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


Level passed: true

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.

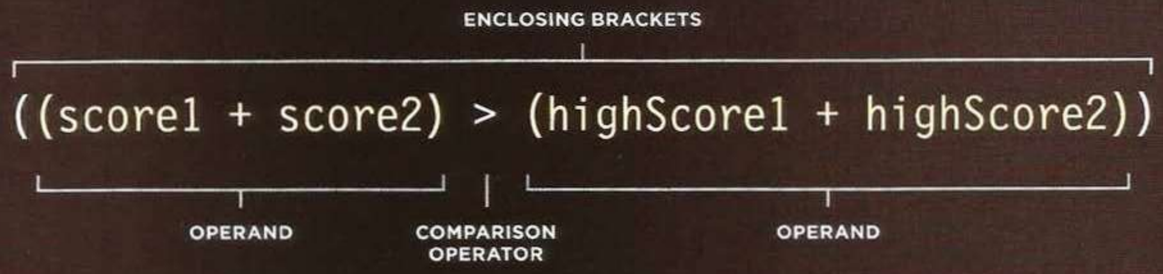
The example starts by setting two variables:
1. `pass` to hold the pass mark
2. `score` to hold the users score

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).



```
((score1 + score2) > (highScore1 + highScore2))
```

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.

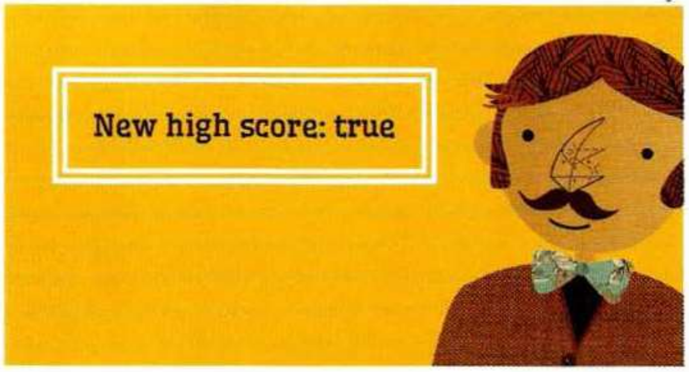
The comparison operator checks if the user's total score is greater than the highest score for the test and stores the result in a variable called `comparison`.

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



New high score: true

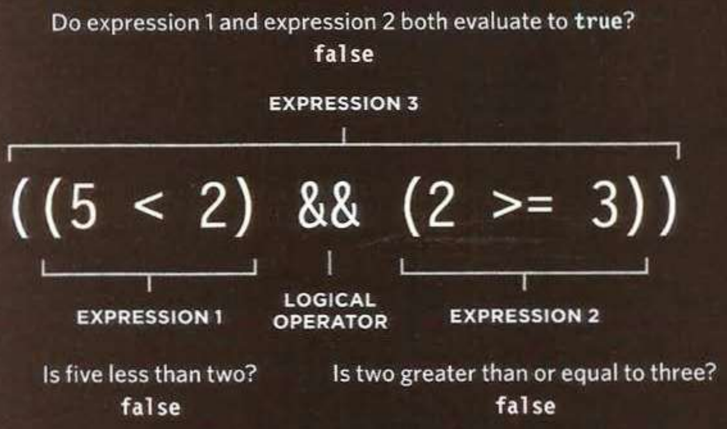
In the comparison operator, the operand on the left calculates the user's total score. The operand on the right adds together the highest scores for each round. The result is then added to the page.

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.



```
((5 < 2) && (2 >= 3))
```

Do expression 1 and expression 2 both evaluate to true?
false
EXPRESSION 3
EXPRESSION 1
LOGICAL OPERATOR
EXPRESSION 2
Is five less than two?
false
Is two greater than or equal to three?
false

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.

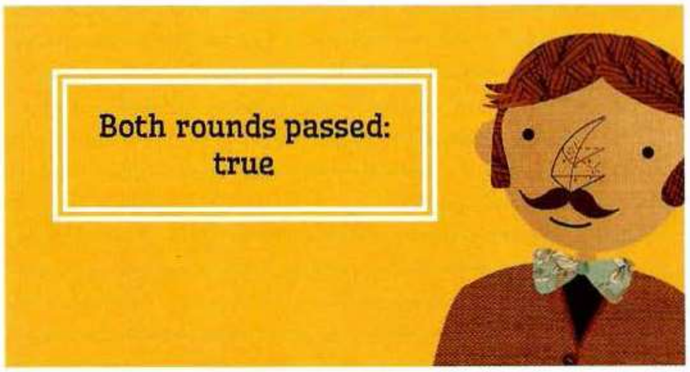
c04/js/logical-and.js

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



Both rounds passed: true

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`.

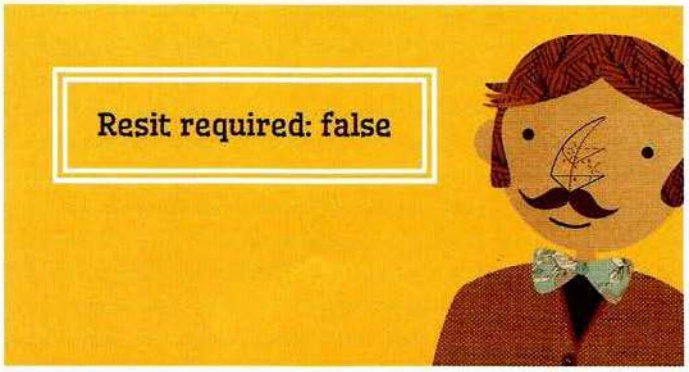
Next, the message is stored in a variable called `msg`. At the end of the message, the logical NOT will invert the result of the Boolean variable so it is `false`. It is then written into the page.

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;
```

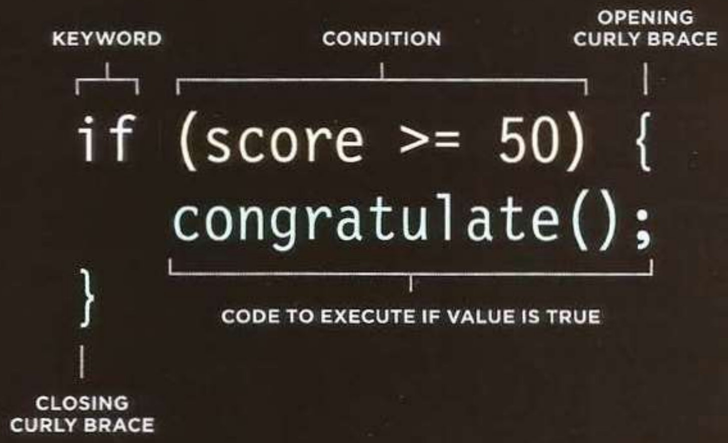
RESULT



Resit required: false

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 (score >= 50) {
    congratulate();
}
```

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

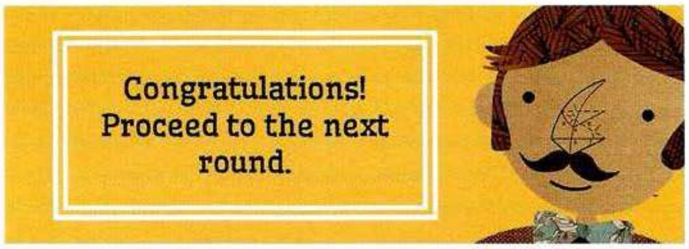
If the condition resolves to **`false`**, the statements in that code block are *not* run. (The script continues to run from the end of the next code block.)

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



Congratulations! Proceed to the next round.

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

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

  ┌function congratulate() {
②┤    msg += 'Congratulations! ';
  └}

    if (score >= 50) {  // If score is 50 or more
①     congratulate();
③     msg += 'Proceed to the next round.';
    }
    var el = document.getElementById('answer');
    el.innerHTML = msg;
```

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.

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;
```

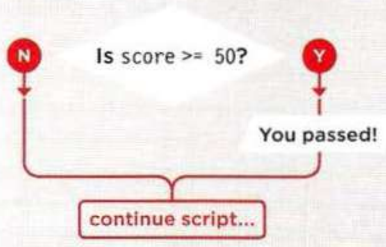
RESULT

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`

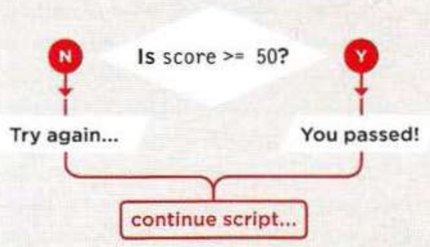
In this test, there are two possible outcomes: a user can either get a score equal to or greater than the pass mark (which means they pass), or they can score less than the pass mark (which means they fail). One response is required for each eventuality. The response is then written to the page.

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`:


N
Is score >= 50?
Y
Try again...
You passed!
continue script...