CONTROL STRUCTURES AND OPERATORS

CI435: Introduction to Web Development

Semester 2

Robin Heath (with thanks to Marcus Winter)

Session overview

- Last week we looked at how to integrate JS with HTML via the `<script>` element, and at JavaScript statements, comments, variables, data types and operators

- This week we’ll look at:
  - quick and dirty input and output
  - control structures
  - comparison operators, logical operators, expressions

- Next week we'll look in more detail at strings

QUICK AND DIRTY INPUT AND OUTPUT

Reading values from HTML

In the HTML page:

```
    <input id="txtGuess" type="text">
```

Get access from JavaScript

```
    var txtGuess = document.querySelector('#txtGuess’);
```

Get the value from JavaScript

```
  var guess = parseInt(txtGuess.value);
```

Writing values to HTML

In the HTML page:

```
    <p id="feedback"></p>
```

Get access from JavaScript

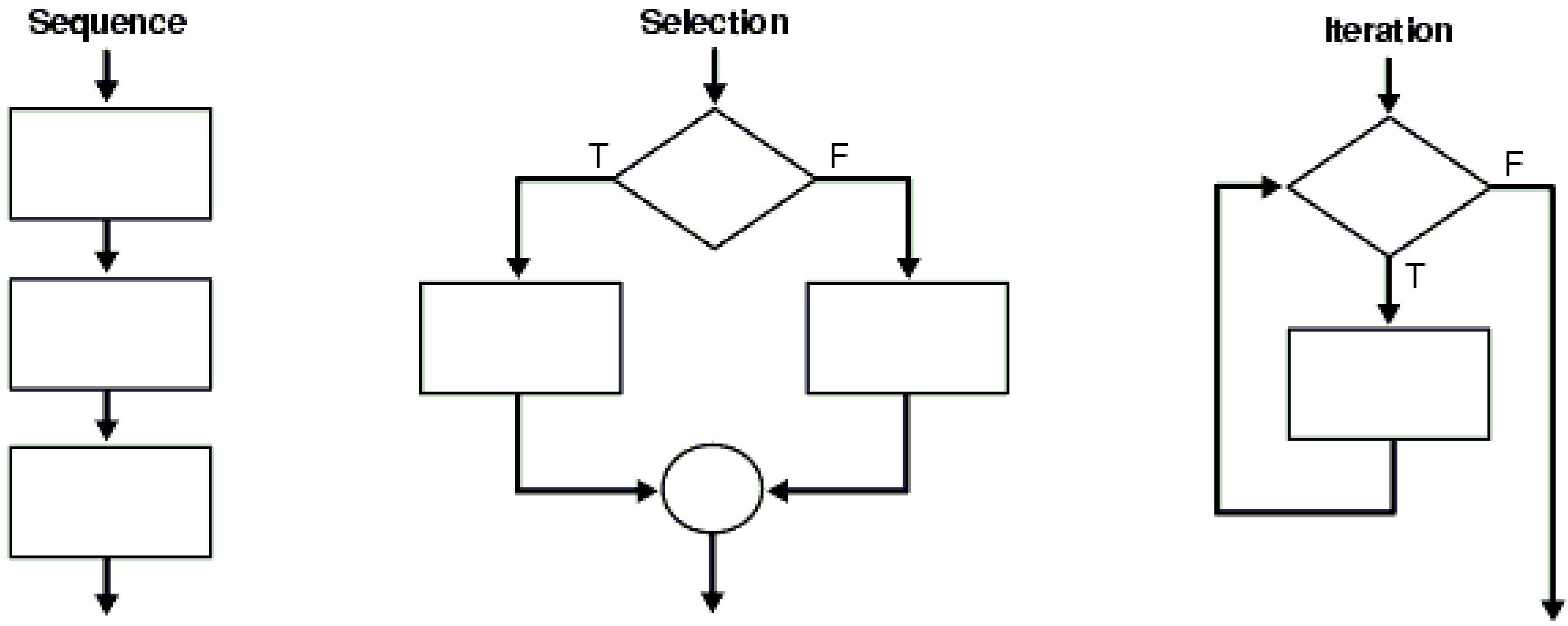
```
    var feedback = document.querySelector('#feedback’);
```

Set the text content from JavaScript

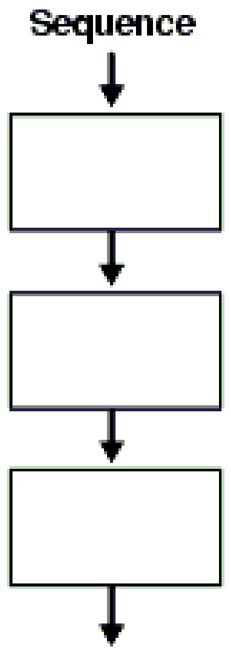
```
  feedback.textContent = 'You got it right';
```

CONTROL STRUCTURES

Control structures



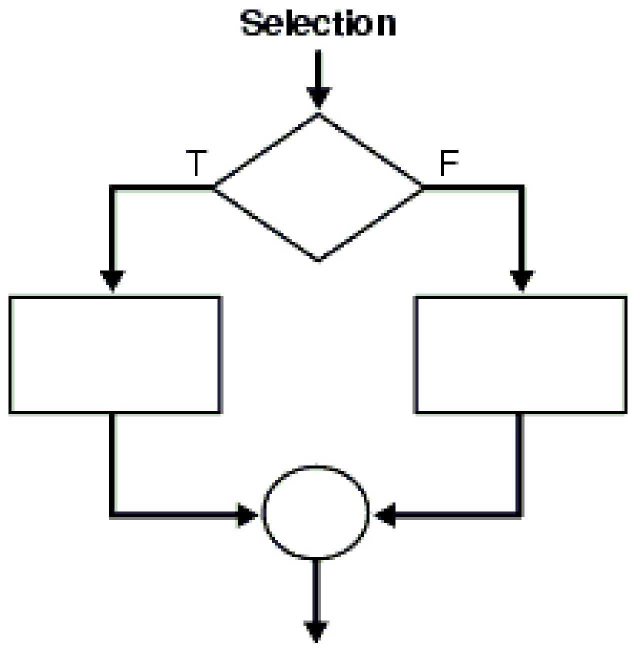
Sequence



Statements are carried out in sequence - one after the other

This is the default behaviour.

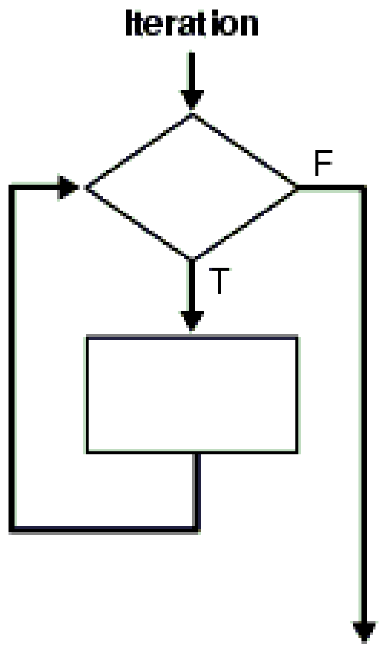
Selection



Alternative statements are carried out depending on a condition.

Branching constructs include `if/else` and `switch/case`

Iteration



Statements are carried out repeatedly while or until a condition is met.

Looping constructs include `while` and `for`

SELECTION: BRANCHING STRUCTURES

Basic `if` statement

One of the most common branching statements:

```
if (condition) {
    // statements for when condition is true
}
```

If the `condition` evaluates to `true` then the statements are executed, otherwise not.

Classic `if / else` statement

Often we have two alternatives depending on a `condition`:

```
if (condition) {
    // statements for when condition is true
} else {
    // statements for when condition is false
}
```

Evaluates `condition` as `true` or `false` and executes statements accordingly.

Multiple `if / else` statements

`if / else` statements can be chained, e.g.

```
var dayOfTheWeek = 2;
if (dayOfTheWeek === 1) {
   console.log("Monday");
} else if (dayOfTheWeek === 2) {
   console.log("Tuesday");
} else if (dayOfTheWeek === 3) {
   console.log("Wednesday");
} // and so on…
```

Switch Statement

Alternative to multiple if-else statements, more suitable if there are many possible conditions:

```
var dayOfTheWeek = 2;
switch(dayOfTheWeek) {
   case 1:
        console.log("Monday");
        break;
   case 2:
        console.log("Tuesday");
        break;
   …
   default:
          console.log("Not a valid day");
}
```

ITERATION:
LOOPING STRUCTURES

Classic `for` loop

Useful when statements need to be repeated for a certain number of times:

```
for(initialize; test; increment) {
   // statements;
}
```

Example:

```
for(var i=0; i<10; i++) {
     console.log(i);
}
```

Variants of the `for` loop

JavaScript has several variants of the classic for loop - some of which are not considered good practice:

```
for each (prop in object) {         // deprecated: do not use
    // statements
}
for (prop in object) {          // typically used with JSON
    // statements
}
for (val of iterable) {         // new in ES6
    // statements
}
array.forEach(function(item){       // specific to arrays
    // statements
});
```

» We'll cover these later when discussing arrays and objects

Classic `while` loop

Useful when statements need to be repeated until a condition is met (but we don't know exactly how often):

```
while (condition) {
    // statements;
}
```

Example:

```
var connected = false;
while(!connected) {
   connected = try_to_connect();
}
```

Less common `do...while` loop

Tests `condition` at the end of the loop rather than the beginning (i.e. first iteration is always executed):

```
do {
   // statement
} while (condition);
```

Example:

```
var connected;
do {
   connected = try_to_connect();
} while (!connected);
```

The `break` statement

The `break` statement can be used to exit a block of statements regardless of the `condition` controlling its execution.

When exiting a block of statements with `break`, then the program execution resumes with the first statement after that block (if there is any).

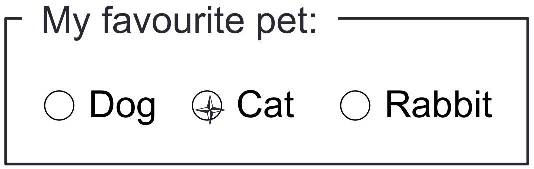
`break` is typically used in `switch` statements and `loops`

Breaking out of a `switch` statement:

```
var dayOfTheWeek = 2;
switch(dayOfTheWeek) {
    case 1:
          console.log("Monday");
          break;
    case 2:
          console.log("Tuesday");
          break;
    …
    default:
          console.log("Not a valid day");
}
```

Breaking out of a `for` loop:

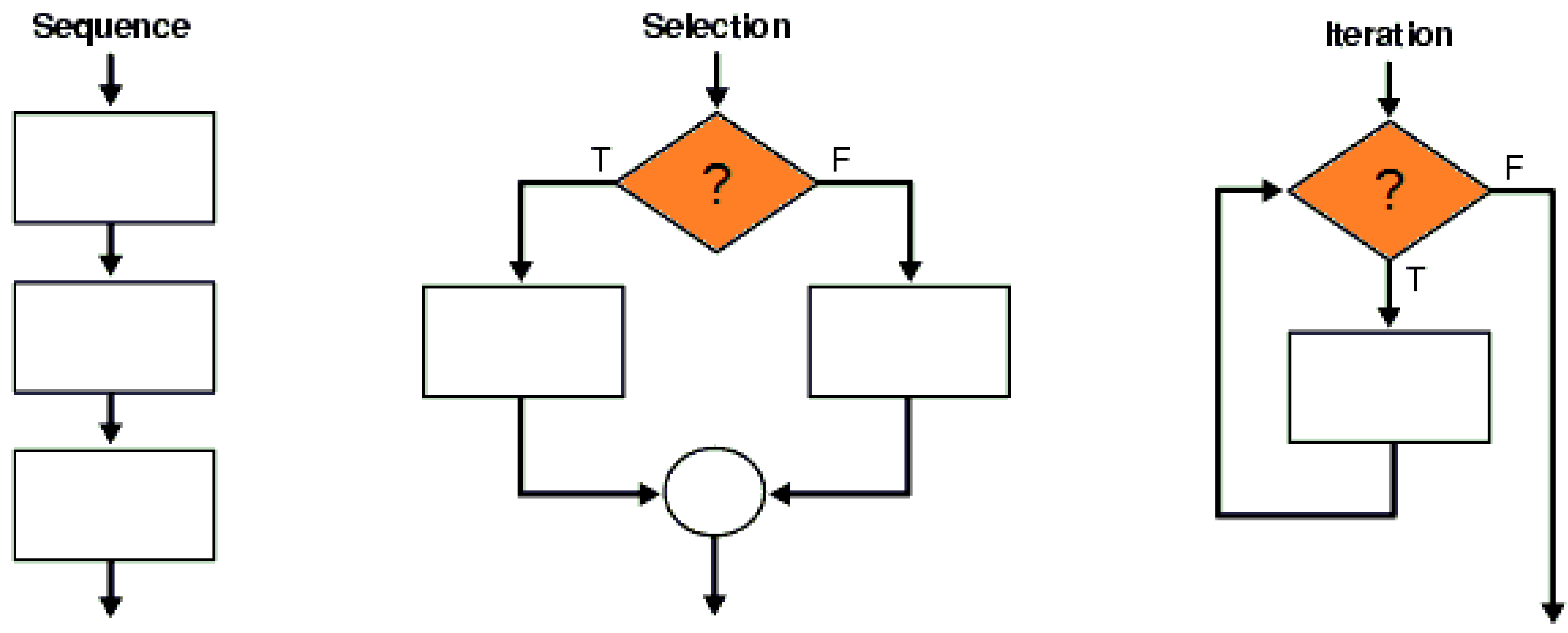
```
// example 1: breaking an out of infinite look
while(true) {
    if(try_to_connect()) {
   break;
    }
}



// example 2
var i, value = null;
for(i = 0; i < radio.length; i++) {
    if(radio[i].checked) {
        value = radio[i].value;
        break;
    }
}
```


My favourite pet:

○ Dog [✓ Cat] ○ Rabbit

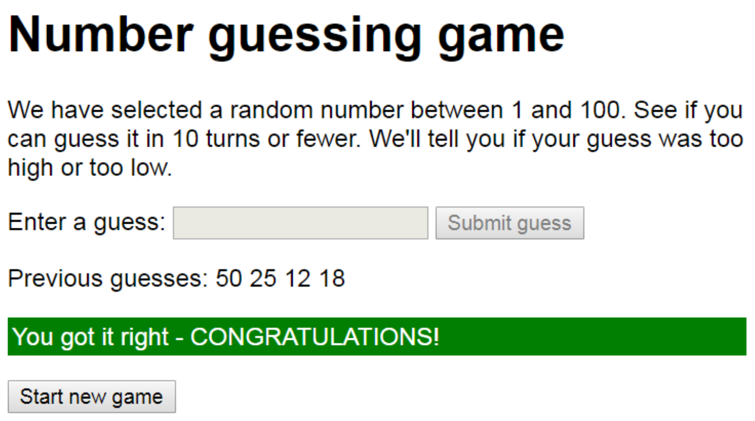
COMPARISON OPERATORS, LOGICAL OPERATORS AND EXPRESSIONS

A closer look at conditions



Example

http://rh37.brighton.domains/ci435/wk1/number_guessing_game/


Number guessing game

We have selected a random number between 1 and 100. See if you can guess it in 10 turns or fewer. We'll tell you if your guess was too high or too low.

Enter a guess:

Previous guesses: 50 25 12 18

You got it right - CONGRATULATIONS!

- Have we made 10 guesses or less?
- Is the guessed number greater, less or equal to our secret number?

Operators and expressions

1. Use comparison operators and/or logical operators to formulate **expressions**.

2. Expressions evaluate to a value, which can be interpreted as **true** or **false**

```
var a = 0;


console.log(a > 0);


if(a > 0) {
   // statement 1
} else {
   // statement 2
}
```

Comparison operators

… used to compare values

`>`	greater than	`>=`	greater than or equal to
`<`	less than	`<=`	less than or equal to
`==`	equal to	`!=`	not equal to (interpreted)
`===`	equal to	`!==`	not equal to (strict)

Comparison operators

Example 1:

```
var a = 1, b = 2;

if(a > b) {
    console.log("a greater than b");
} else if(a < b) {
    console.log("a less than b");
} else {
    console.log("a equals b");
}
```

What will be printed to the console?

Comparison operators

Example 1:

```
var a = 1, b = 2;

if(a > b) {
    console.log("a greater than b");
} else if(a < b) {
    console.log("a less than b");
} else {
    console.log("a equals b");
}
```

Prints "a less than b" to the console

Testing for equality

JS has two types of equality operators:

1. With coercion: `==` and its negation `!=`

   If arguments are not of the same data type, they are first converted to the same data type and then compared, i.e. arguments are equal if they have the **same value** after the conversion

2. Without coercion: `===` and its negation `!==`

   Compares arguments without conversion, i.e. arguments are equal if they have the **same type** and the **same value**

Interesting examples

Bad:

```
''  == '0'        // false
0   == ''         // true
0   == '0'        // true

false == 'false'    // false
false == '0'        // true
```

Good:

```
''  === '0'       // false
0   === ''        // false
0   === '0'       // false

false === 'false'  // false
false === '0'      // false
```

Best practice is to use `===` and `!==` for testing equality, to stay in control and avoid unexpected results

Logical operators

Multiple conditions can be combined (or negated) with logical operators:

`&&` Logical AND

- True if both operands are true, false otherwise
- If the first condition is false the second is not even evaluated

`||` Logical OR

- True if either operand (or both) is true, false otherwise
- If the first condition is true the second condition is not even evaluated

`!` Logical NOT

- Inverts the logical value of its operand

Examples

```
var max = 3, count = 0, connected = false, speed = 20;

(count < max && connected === false)      // ?
(count < max && !connected)             // ?

(count >= max || connected === true)      // ?
(count >= max || connected)             // ?
(count >= max || !connected)            // ?

(!!connected)                    // ?

((connected && speed < 20 && count < max)    // ?
    || (!connected && count < max))
```

Examples

```
var max = 3, count = 0, connected = false, speed = 20;

(count < max && connected === false)      // true
(count < max && !connected)            // true

(count >= max || connected === true)      // false
(count >= max || connected)            // false
(count >= max || !connected)           // true

(!!connected)                 // false

((connected && speed < 20 && count < max)     // true
    || (!connected && count < max))
```

Conditional (ternary) operator

- Only operator that uses three operands
- Typically used as a shorthand for `if / else`

```
condition ? expr1 : expr2
```

Example:

```
var isMember = checkMembership(id);    // true

var entry_fee = isMember ? '£5.80' : '£10.20';

console.log(entry_fee);             // ?
```

Recommended reading

Making decisions in your code - conditionals
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/conditionals

Looping code
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Looping_code

Comparison operators
https://developer.mozilla.org/bm/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

Logical operators
https://developer.mozilla.org/bm/docs/Web/JavaScript/Reference/Operators/Logical_Operators

Conditional (ternary) operator
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

KEEP
CALM
AND
KEEP
CODING