# Programming 1

## Javascript

## Lecture #1: Introduction to Javascript

# What is Javascript?

- Javascript is NOT Java!
  - Originally created by Netscape (RIP ☺)
  - …it was originally going to be called LiveScript, but Java was fashionable at the time, so Netscape decided to jump on the bandwagon and call it Javascript
  - There are similarities between Java and Javascript
    - Both are C like languages, so use similar constructs for **`for`**, **`while`**, **`do/while`**, **`if`** etc.
  - There are differences between Java and Javascript
    - Java is *strongly typed*, Javascript is *weakly typed*
    - Java is compiled*, Javascript is interpreted
    - Java is an out-and-out object oriented language, Javascript supports many ways of programming (including OO)

# Javascript in (web) practice

- Mostly (but not exclusively) used as a client-side scripting language for the web
- Javascript is built into all modern web browsers
- Javascript code is either
  - embedded into an HTML page in a `<script>` element
  - placed in an external file which will be referenced in the HTML page
- Either way, to put it simply, Javascript code is sent by a web server alongside the HTML of the page
- The browser then runs the Javascript code – the code runs *client side*.

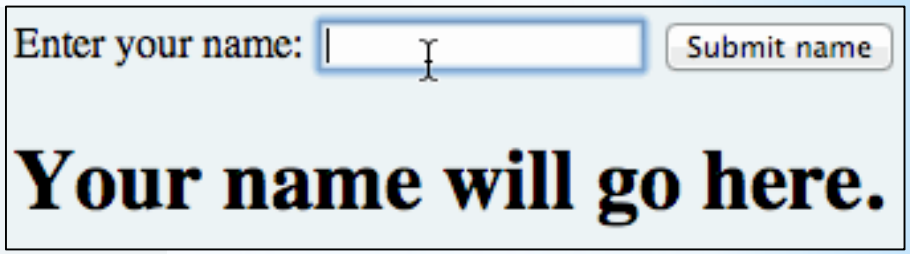
# The relationship between Javascript and HTML

- Ordinarily, a Javascript program will be written to interact with an accompanying HTML page
  - There is no direct equivalent of **`DISPLAY`** (from Banana) or **`System.out.println`** (from Java)
  - There is no direct equivalent of **`GET`** (from Banana) or the **`Scanner`** from Java
- Translation, in English: there's no direct way of getting user input or displaying something to the user in Javascript
  - To get input from the user, a Javascript program might read the contents of an HTML form field
  - To display output to the user, a Javascript program might modify the contents of an HTML element or elements

# The relationship between Javascript and HTML

```html
<!doctype html>
<html>
  <head>
    <script src="jscode.js"></script>
  </head>
  <body>
    <p>
      Enter your name:
      <input id="nameinput"/>
      <button onclick="getName()">Submit name</button>
    </p>
    <h1 id="result">Your name will go here.</h1>
  </body>
</html>
```

HTML




```javascript
function getName()
{
        var nameEl = document.getElementById("nameinput");
        var resultEl = document.getElementById("result");
        var inputtedName = nameEl.value;
        resultEl.innerHTML = inputtedName;
}
```

Javascript

# However… to make things easier in the beginning…

- In the first couple of weeks, we want you to get used to the Javascript language and not have to worry about the complexities of HTML's involvement as well
- So, we will give you a console for input and output
  - You can use `console.log` to print something out to this console
  - You can use `prompt` to get input from the user
- In week three of this unit, we'll draw back the curtain and remove this support, and you'll start writing "proper" Javascript in the same way it's used on real web pages

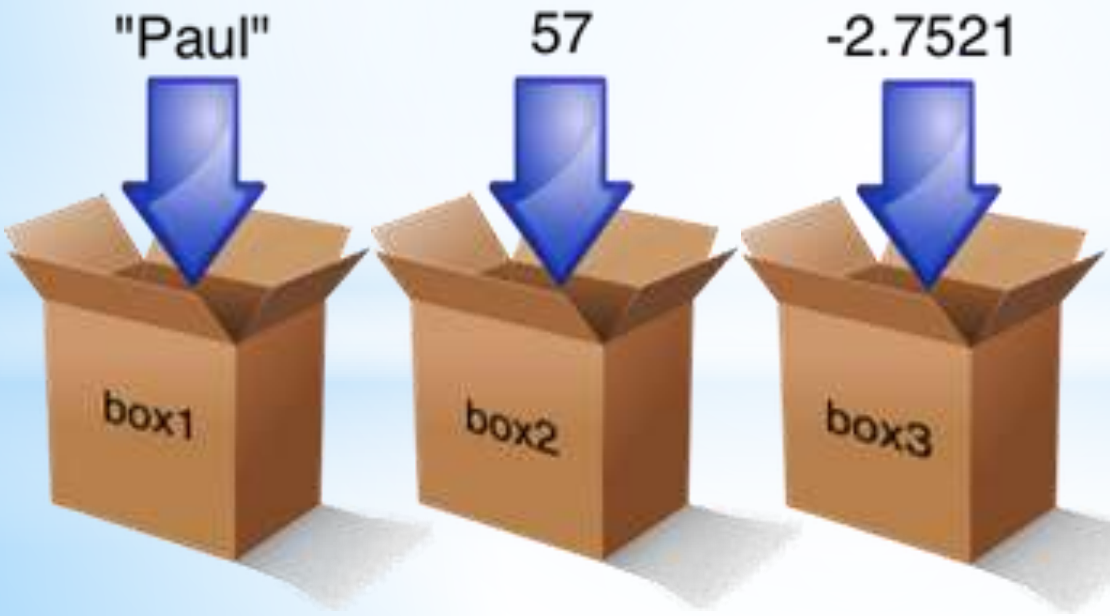
# Javascript – the basics of the language

# Javascript is a C-like language

- Most of the basic constructs in Javascript such as **`for`**, **`while`**, **`do/while`** and **`if`** are similar to Java and/or PHP
- Be sure to remember the role of code blocks { } and semicolons, and when these are necessary!
- Javascript is *weakly typed* – more about this later!

# Variables - a recap

- A variable is like a box – although with certain rules and restrictions
  - Our boxes have labels on the side
  - We can put one thing only into each box




In Javascript, this would be

```
var box1 = "Paul";
var box2 = 57;
var box3 = -2.7521;
```

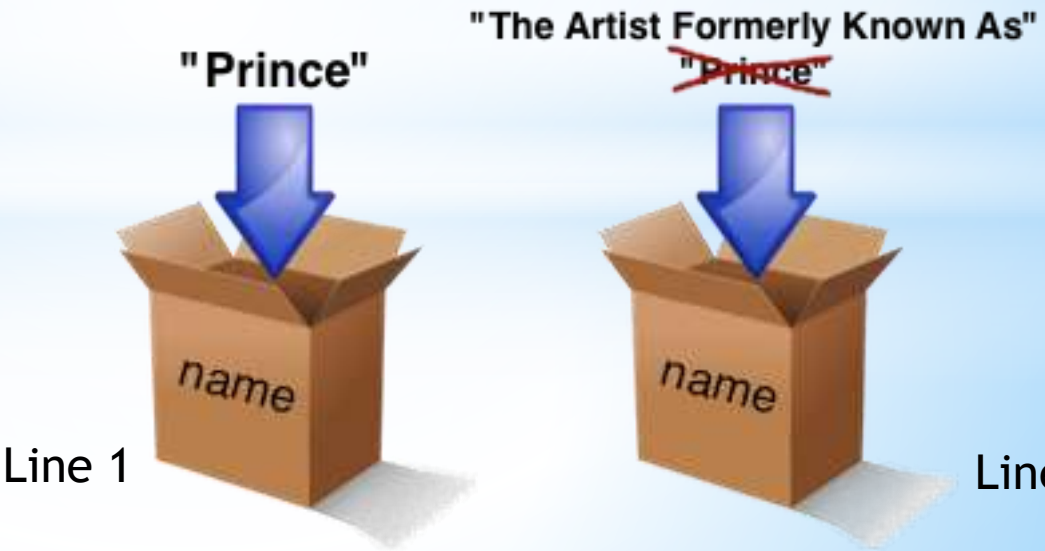
In Banana, this would be

```
set box1 = "Paul"
set box2 = 57
set box3 = -2.7521
```

# Variables - a recap

- If you try to put a value into a variable that is already defined, and already contains a value, the new value **replaces** the old one

```
var name = "Prince";
console.log(name);
name = "The Artist Formerly Known As";
console.log(name);
```



# The `var` keyword

- The **`var`** keyword is used to declare a variable in Javascript
- When you've declared it, you don't need to keep repeating **`var`**:

```
var name = "Prince";
```

**`var`** used to declare variable the first time

```
console.log(name);
```

no **`var`** when just accessing (using) the variable

```
name = "The Artist Formerly Known As";
```

no **`var`** when assigning a new value

```
console.log(name);
```

no **`var`** when just accessing (using) the variable

# Variables - a recap

- You can use a variable in code anywhere where the value it contains would "fit"
  - e.g.

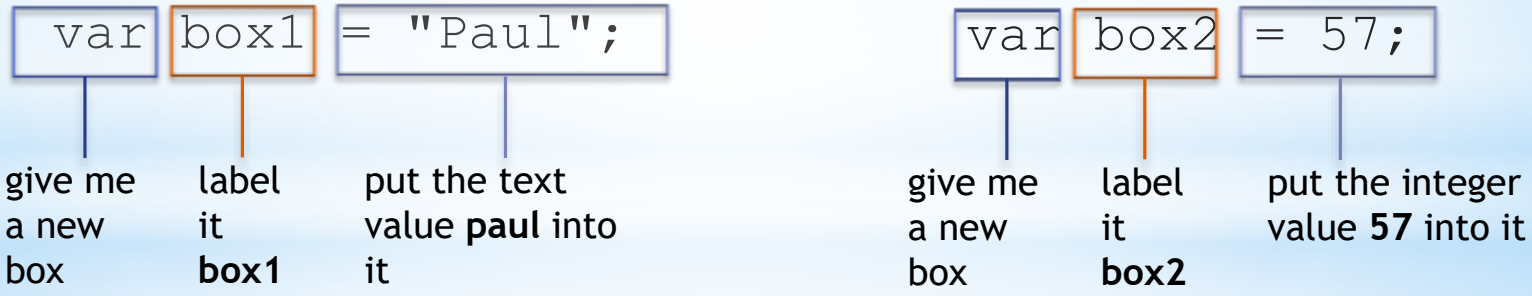
```
var myVariable = 2;
console.log(myVariable);
console.log(myVariable+6);
```

  - is *grammatically* the same as

```
console.log(2);
console.log(2+6);
```

  - "REMEMBER THE GRAMMAR"...

# Variables in Javascript: the basics

- Javascript is a **weakly typed** language
  - This means that you do NOT have to specify the type of data that you store in each variable (unlike Java, which is strongly typed)
  - When you indicate that you are declaring a new variable with the keyword `var` that's all you need - no
  - 

  - Javascript will automatically "figure out" what kind of data you're putting into the variable

# Variables - a recap

- You can use a variable in code anywhere where the value it contains would "fit"
  - e.g.

```
var anotherVariable = "Good morning";
console.log(myVariable);
```

  - is basically saying

```
var anotherVariable = "Good morning";
console.log(~~myVariable~~);
```

**"Good morning"**

  - but don't make the mistake

```
var anotherVariable = "Good morning";
console.log("myVariable");
```

..unless of course your intention is actually to print the text **myVariable** and NOT the contents of the variable itself!

# Variables in Javascript: it's forgiving

- You don't *have* to declare a new variable with the `var` keyword, but you *should*

```
var stuff = "some text";
console.log(stuff);
stuff = "new text";
console.log(stuff);
```

```
stuff = "some text"
console.log(stuff);
stuff = "new text";
console.log(stuff);
```

- You can get away with re-declaring your variable every time, but you *shouldn't*

```
var stuff = "some text";
console.log(stuff);
stuff = "new text";
console.log(stuff);
```

```
var stuff = "some text"
console.log(stuff);
var stuff = "new text";
console.log(stuff);
```

# Numeric operators in Javascript

- All the usual ones you'll have seen in Banana

  | | |
  |---|---|
  | + | add (or concatenate* when a string/text value is involved) |
  | - | Subtract |
  | * | multiply |
  | / | divide |
  | % | remainder (modulus) |

- So
  - 4 + 2 = 6
  - 4 - 2 = 2
  - 4 * 2 = 8
  - 13 % 3 = 1 (13 divided by 3 is 2 with a remainder of 1)

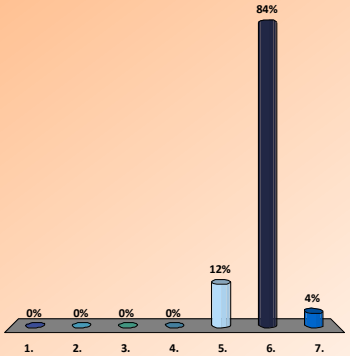
- "abc" + "def" = "abcdef"
- 27 + "xyz" = "27xyz"

# Variables and doing calculations

```
var age = 39;
age = age + 1; // another year older ☹

var age = 27;
age++;              // shorthand for age = age + 1
                    // there's also age-- to deduct by 1

var coursework = 25;
var exam = 46;
var totalMark = coursework + exam;
var totalPercent = totalMark / 80 * 100;
```

# What will be the output of the following code?

```
var box1 = "Paul";
var box2 = 57;
var box3 = -2.7521;
box2 = box2 + 10;
box1 = box2 + box1
console.log(box1)
```




1. 67
2. 57
3. 7.2479
4. 7.2479Paul
5. 57Paul
6. 67Paul
7. There would be an error

# The C-Like IF statement

- Banana

```
set number = 10
if number > 5
   display "Number is big"
endif
```

- Javascript

```
var number = 10;
if (number > 5)
{
  console.log("Number is big");
}
```

# The C-Like IF statement

- Banana

```
set number = 10
if number > 5
   display "Number is big"
endif
```

- Javascript

```
var number = 10;
if (number > 5)
{
  console.log("Number is big");
}
```

- Most C-like languages don't use explicit keywords like **`endif`** or **`endfor`** to mark the end of a construct
- Instead, curly brackets { } are used to denote a *code block*
  - The start of a code block is the left facing curly bracket {
  - The end of a code block is the right facing curly bracket }
- The contents of a code block go with the preceding statement
- So in this case, the code **`console.log("Number is big")`** within the code block gets run when the condition in the **`if`** statement is true

# Conditions in conditional statements

- Banana

```
set number = 10
if number > 5
   display "Number is big"
endif
```

- Javascript

```
var number = 10;
if (number > 5)
{
  console.log("Number is big");
}
```

- The conditional part of a conditional statement should be enclosed in brackets
- This includes **`if`** statements, **`while`** statements and **`do/while`** statements (equivalent to repeat/until in Banana)

# Semi-colons

- Javascript

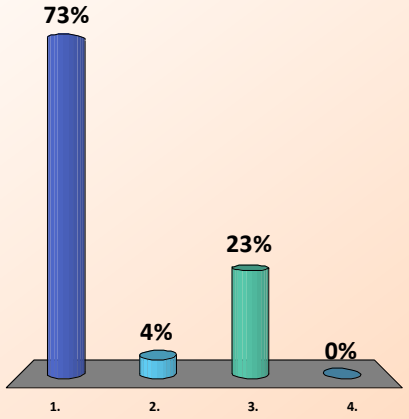
```
var number = 10;
if (number > 5)
{
  console.log("Number is big");
}
```

- Semi-colons denote the end of a statement
- Think of them like being like a full-stop at the end of a sentence in English
- You MUST have a semi-colon after every statement
  - The one exception is immediately after a curly bracket closing a code block

# What will be the output of this code?

```
var number = 2;
if (number > 5);
{
  console.log("Number is big");
}
```

1. It will print nothing (but won't cause an error)
2. It will print **`Number is big`**
3. It will print nothing other than an error
4. It will print the value of the variable **`number`** (i.e. 2)



# The C-Like IF statement and semi-colons

- A common mistake:

```
var number = 2;
if (number > 5) ;
{
  console.log("Number is big");
}
```

- You should NEVER ever ever have a semi-colon immediately after the condition of an `if` statement
- A semi-colon signifies the end of a statement
  - An if statement doesn't end after the condition!
  - It ends at the closing curly bracket of the associated code block

# The C-Like IF statement and semi-colons

- A common mistake:

```
var number = 2;
if (number > 5) ;
{
  console.log("Number is big");
}
```

- So here you are actually saying
- "if number is greater than 5"…

# The C-Like IF statement and semi-colons

- A common mistake:

```
var number = 2;
if (number > 5) ;
{
  console.log("Number is big");
}
```

- So here you are actually saying
- "if number is greater than 5"…
- …do this (i.e. do *nothing*!)

# The C-Like IF statement and semi-colons

- A common mistake:

```
var number = 2;
if  (number > 5);
{
  console.log("Number is big");
}
```

- So here you are actually saying
- "if number is greater than 5"…
- …do this (i.e. do *nothing*!)
- The now-orphaned code block would *always* execute
  - it is not associated with the `if` statement
  - the `if` statement finishes at the semi-colon

# The C-Like IF statement and semi-colons

- Wrong

```
var number = 7;
if  (number < 5);
{
  console.log("Number is small");
}
```

- Right

```
var number = 7;
if  (number < 5)
{
  console.log("Number is small");
}
```

- What would the code on the left print?
- What would the code on the right print?

# The C-Like IF statement and semi-colons

- Wrong

```
var number = 7;
if  (number < 5) ;
{
  console.log("Number is small");
}
```

- *if number is less than 5 do this…*
- …after you've checked, carry on and run this code

- Right

```
var number = 7;
if  (number < 5)
{
  console.log("Number is small");
}
```

- *if number is less than 5 do this…*

# The C-Like IF/ELSE statement

- Banana

```
set number = 10
if number > 5
   display "Number is big"
else
   display "Number is small"
endif
```

- Javascript

```
var number = 10;
if (number > 5)
{
  console.log("Number is big");
}
else
{
  console.log("Number is small");
}
```

# The C-Like IF/ELSE statement

- Banana

```
set number = 10
if number > 5
   display "Number is big"
else
   display "Number is small"
endif
```

- Javascript

```
var number = 10;
if (number > 5)
{
  console.log("Number is big");
}
else
{
  console.log("Number is small");
}
```

- else statements have their own, discrete code block
- Unlike in our Banana, the **`else`** keyword does not by itself denote the end of the initial **`if`** condition

# The C-Like IF/ELSE statement

- Correct

```
var number = 10;
if (number > 5)
{
  console.log("Number is big");
}
else
{
  console.log("Number is small");
}
```

- Incorrect

```
var number = 10;
if (number > 5)
{
  console.log("Number is big");
else
  console.log("Number is small");
}
```

# The C-Like IF/ELSE statement

- Valid

```
var number = 10;
if (number > 5)
{
  console.log("Number is big");
}
else
{
  console.log("Number is small");
}
```

- Also valid

```
var number = 10;
if (number > 5) {
  console.log("Number is big");
}
else {
  console.log("Number is small");
}
```

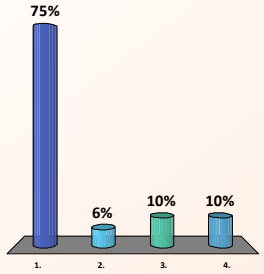
- Also also valid (but really not advised! ☺)

```
var number = 10; if (number > 5) { console.log("Number is big"); } else
{  console.log("Number is small"); }
```

# What would the code below print?

```
var number = 2.185;
if (number > 2)
{
    console.log("Not two bad…");
}
else if (number < 3)
{
    console.log("Threedom!");
}
else
{
    console.log("I'm confused");
}
```

1. *Not two bad…*
2. *Threedom!*
3. *I'm confused*
4. There would be an error



# The C-Like FOR loop

```
for (var i = 0; i <= 4; i = i + 1)
{
   console.log("Programming is awesome");
}
```

equivalent to the Banana

```
FOR i = 0 TO 4
DISPLAY "Programming is awesome"
ENDFOR
```

# The C-Like FOR loop

```
for (var i = 0; i <= 4; i = i + 1)
{
    console.log("You are awesome!");
}
```

equivalent to the Banana

```
FOR I = 0 TO 4
DISPLAY "You are awesome!"
ENDFOR
```

Declares and initialises the counter variable – in this case, the counter is *i* and it starts at zero.

Specifies the *condition* under which the loop continues to repeat – in this case it keeps going WHILE it's less or equal to 4.

What happens to the counter variable every time we repeat the loop – in this case, it is increased by 1.

# The C-Like FOR loop

```
for (var i = 0; i <= 4; i = i + 1)
{
   console.log("Programming is awesome");
}
```

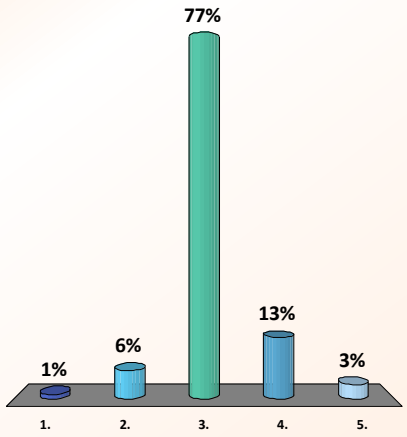
would more commonly be written

```
for (var i = 0; i < 5; i++)
{
   console.log("Programming is awesome");
}
```

# What will the program below output?

```
for (var i = 10; i >= 4; i = i - 1)
{
   console.log("i");
}
```

1. It would print the numbers from 4 to 10
2. It would print the numbers from 4 to 10 backward
3. It would print “i” 7 times
4. It would print “I” 6 times
5. There would be an error



# The C-Like WHILE loop

```
var i = 0;
while ( i < 5 )
{
   console.log("You are awesome!");
   i++;
}
```

equivalent to the Banana

```
SET i = 0;
WHILE i < 5
  DISPLAY "You are awesome!"
  i = i + 1;
ENDWHILE
```

# The C-Like WHILE loop

```
var i = 0;
while (i < 5)
{
   console.log("You are awesome!");
   i++;
}
```

equivalent to the Banana

```
SET i = 0;
WHILE i < 5
  DISPLAY "You are awesome!"
  i = i + 1;
ENDWHILE
```

The **test condition** specifies the *condition* under which the loop continues to repeat – in this case it keeps going WHILE the variable i is less than 5.

While the **test condition** evaluates to true, these statements repeatedly run.

# The C-like DO/WHILE loop

- In our Banana, we've used **`repeat/until`** as our *post-test* loop
  - this means it tests the condition AFTER each repetition
  - contrast **`while/endwhile`**, a *pre-test* loop, which tests the condition BEFORE each repetition
- C-like languages do not have a direct equivalent to **`repeat/until`**
- Instead, **`do/while`** can be used as a post-test loop
- There is one crucial difference, however…

# The C-like DO/WHILE loop

```
var i = 0;
do
{
   console.log("You are awesome");
   i++;
} while (i != 5);
```

equivalent to the Banana

```
SET i = 0;
REPEAT
  DISPLAY "You is awesome"
  i = i + 1;
UNTIL i == 5
```

# The C-like DO/WHILE loop

The **test expression** specifies the *condition* under which the loop continues to repeat – in this case it keeps going WHILE the variable i is not equal to 5.

```
var i = 0;
do
{
   console.log("You are awesome");
   i++;
} while (i != 5);
```

equivalent to the Banana

```
SET i = 0;
REPEAT
  DISPLAY "You is awesome"
  i = i + 1;
UNTIL i == 5
```

While the **test expression** evaluates to true, these statements while repeatedly run.

**IMPORTANT:** note the fundamental difference between the REPEAT/UNTIL construct you may have seen in Banana, and the C-like DO/WHILE.

- REPEAT/UNTIL repeats **until** the final condition is TRUE.
  - **So when it is true it stops!**
- DO/WHILE repeats **while** the condition is true
  - **So when it is true it repeats!**

# Relational operators

- Usually used in conditional statements e.g. IF or WHILE

```
var speed = 19;
if (speed >= 45)
{
   console.log("Too fast");
}
else if (speed >= 25)
{
   console.log("Just right!");
}
else
{
   console.log("Too slow!");
}
```

**What message will be printed?**

# Relational operators

- Usually used in conditional statements e.g. IF or WHILE

```
var speed = 19;
if (speed >= 45) false
{
   console.log("Too fast");
}
else if (speed >= 25) false
{
   console.log("Just right!");
}
else
{
   console.log("Too slow!");
}
```

# Relational operators

- Usually used in conditional statements e.g. IF or WHILE

```
var speed = 35;
if (speed >= 45)
{
   console.log("Too fast");
}
else if (speed >= 25)
{
   console.log("Just right!");
}
else
{
   console.log("Too slow!");
}
```

# Relational operators

- Usually used in conditional statements e.g. IF or WHILE

```
var speed = 35;
if (speed >= 45)
{
   console.log("Too fast");
}
else if (speed >= 25) true
{
   console.log("Just right!");
}
else (else does not run if condition is true)
{
   console.log("Too slow!");
}
```

# Relational operators

- Usually used in conditional statements e.g. IF or WHILE

```
var x = 2;
while (x < 500)
{
  console.log(x);
  x = x * 2;
}

if (x > 510)
{
  console.log("Final result was more than 510");
}
```

# Logical operators

| | | |
|---|---|---|
| \|\| | OR | (one condition must be true) |
| && | AND | (ALL conditions must be true) |

- Use these to "join" relational operators together… e.g.

```
var number = prompt("Type a number");
if (number >=5 && number <= 10)
{
   console.log("Number was between 5 and 10 inclusive");
}
else if (number > 10 || number < 1)
{
   console.log("Number was outside the accepted range of 1-10");
}
else
{
   console.log("Number was between 1 and 4");
}
```

# NOT

- Use the ! to invert a boolean expression
  - …if it's currently true, it becomes false
  - …if currently false, it becomes true
- This is called "NOT"

```
var number = prompt("Type a number");
if (!(number >= 1 && number <= 10))
{
   console.log("Number was outside 1-10");
}
```

# NOT

- …assuming we typed 17…

```
var number = prompt("Type a number");
if (!(number >= 1 && number <= 10))
{


   console.log("Number was outside 1-10");
}
```

# NOT

- …assuming we typed 17…

```
var number = prompt("Type a number");
if (!(number >= 1 && number <= 10))
{          true      &&      false


   console.log("Number was outside 1-10");
}
```

# NOT

- …assuming we typed 17…

```
var number = prompt("Type a number");
if (!(number >= 1 && number <= 10))
{       true    &&    false
    ! (          false          )

   console.log("Number was outside 1-10");
}
```

# NOT

- …assuming we typed 17…

```
var number = prompt("Type a number");
if (!(number >= 1 && number <= 10))
{        true      &&      false
    ! (           false            )
                    true
   console.log("Number was outside 1-10");
}
```

# Refreshing your memory: C-Like/Javascript `if` statement

- If statement

```
if (some condition)
{
     do something…
}
else
{
    do something else…
}
```

# Refreshing your memory: Javascript `if` statement

- If statement

```
var text = prompt("Type something");
if (text == "abc")
{
  console.log("You typed abc");
}
else
{
  console.log("You typed something else");
}
```

# Refreshing your memory: Javascript `if` statement

- If statement

```
var text = prompt("Type something");
if ( text == "abc" )
{
  console.log("You typed abc");
}
else
{
  console.log("You typed something else");
}
```

For Java'ers - no need for **`.equals`** to test for string equality in Javascript

# Conditional loops

- Loops while a given condition is true
  - do…while

```
do
{
    ..something to do..
} while (some condition)
```

do/while will ALWAYS execute at least once

  - while

```
while (some condition)
{
   ..something to do..
}
```

while, potentially, might never execute at all (it depends on the condition)

# The FOR loop - a retrospective

- So the FOR loop is basically a composite of different statements:

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

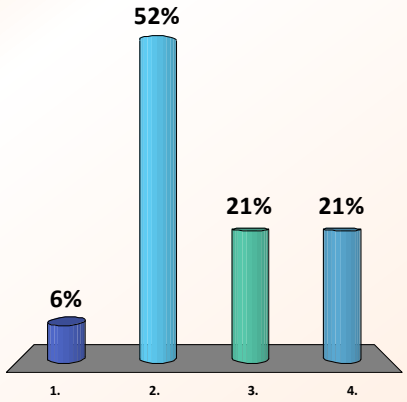
```
var count = 0;
while (count < 10)
{
  console.log(count);
  count++;
}
```

# OK, smartypants. What about this one then?

```
for (var i = "X"; i != "XXXX"; i = i + "X")
{
    console.log("X");
}
```

1. Seriously? Of course that wouldn't run. There'd be an error.
2. It would print "X" 3 times
3. It would print "X" 4 times
4. It would print a little triangle of "X"es, like this:

   X
   XX
   XXX



# Variables in Javascript: it's forgiving, it tries to figure out what you mean - but can make mistakes

- Because Javascript is weakly typed it will try to make sense of things if you mix different types of data:

```
var numberInAString = "27";
numberInAString = numberInAString - 7;
console.log(numberInAString);
```

  - …prints **20** to the console
  - numberInAString now contains a numeric value!

```
var numberInAString = "27";
numberInAString = numberInAString + 7;
console.log(numberInAString);
```

  - …prints **277** to the console… **WHY**?

# Arrays in Javascript

- If a variable is like a box, then an array is like a box with compartments:


…and so on up to the size of the array…

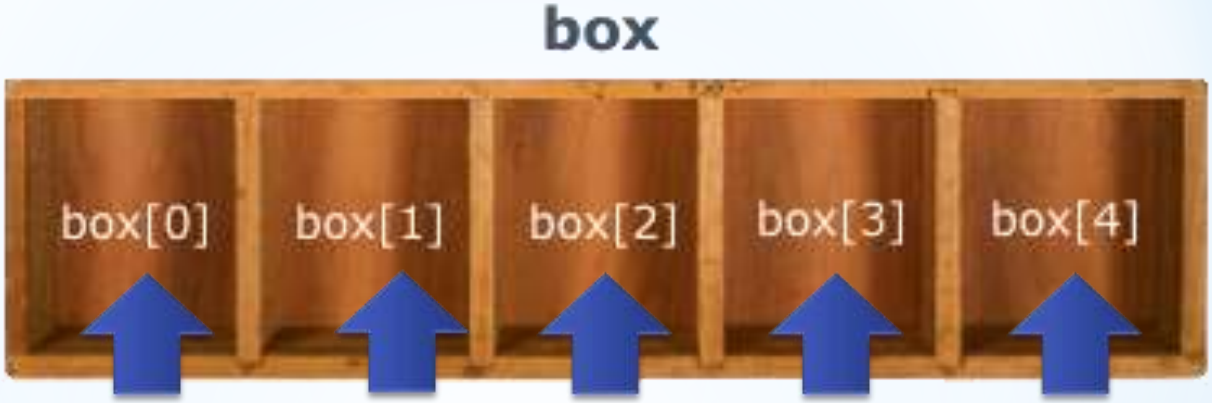
- Several ways you can declare an array in Javascript, but we recommend:

```
var box = [];
```

  - or, if you know what you want to put in the array

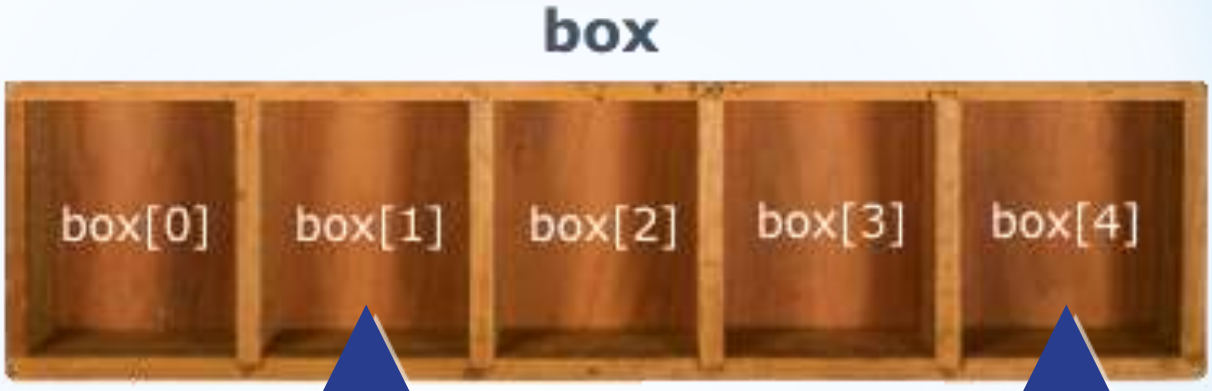
```
var box = ["words","go","in","the","array"];
```

# Arrays in Javascript




```
var box = ["words","go","in","the","array"];
console.log(box[2]); // would print "in"
console.log(box[4]); // would print "array"
```

# Arrays in Javascript




```
var box = [];
box[1] = "foo";
box[4] = "bar";
console.log(box[1]); // would print "foo"
console.log(box[4]); // would print "bar"
console.log(box[2]); // ??? - you tell me! ☺
```

# Arrays in Javascript - forgiving, again

- Javascript arrays do not have to have their length specified when declared - **they will grow as necessary**
- They will also add elements that contain *undefined* as necessary
- You can get the length of an array by adding `.length`

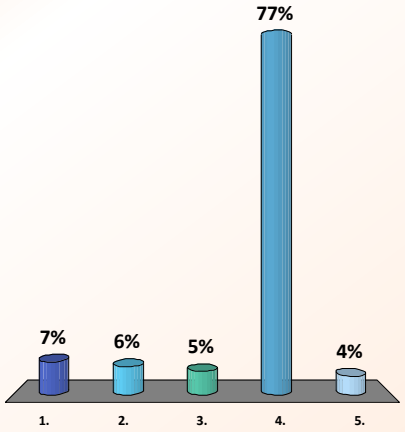
```
var arr = [1,2,3,4];
console.log(arr.length); // prints 4

var arr2 = [];
console.log(arr.length); // prints 0
arr2[2] = "xyz";
console.log(arr.length); // prints 3 – why?
```

# Looping through an array

- By now you *should* be able to compose this code yourself, but…

```
var stuff = ["All","work","and","no","play"];
for (var count = 0; count < stuff.length; count++)
{
   console.log(stuff[count]);
}
```

- What will this do?
  - Think about the *grammar* – what values are in the various variables, what does `stuff.length` result in?




1. It would count from 0 to 4
2. It would print the word All 5 times
3. It would print the word Play 5 times
4. It would print out all the words in the array
5. There would be an error

# Slicing and dicing an array

- If we set up an array…

```
var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
             fruits[0]  fruits[1]  fruits[2] fruits[3]  fruits[4]
```

- …we can use various methods to take subsets of the array
  - **slice**

```
var citrus = fruits.slice(1,3);
```

    - "give me everything starting at fruits[1] UP TO BUT NOT INCLUDING fruits[3]"
    - citrus would contain `Orange` and `Lemon`
    - **fruits is unchanged**

# Slicing and dicing an array

- If we set up an array…

```
var fruits = ["Banana", "Orange", "Lemon",   "Apple", "Mango"];
             fruits[0]   fruits[1]  fruits[2]   fruits[3]  fruits[4]
             fruits[-5]  fruits[-4] fruits[-3]  fruits[-2] fruits[-1]
```

- Use negative numbers with slice to select from the end of the array:

```
var citrus = fruits.slice(-4,-2);
```

  - "give me everything starting at the fourth from the right UP TO BUT NOT INCLUDING the second from the right
  - again, citrus would contain `Orange` and `Lemon`
  - **fruits is still unchanged**

# Slicing and dicing an array

- If we set up an array…

```
var fruits = ["Banana", "Orange", "Lemon",   "Apple", "Mango"];
               fruits[0]   fruits[1]  fruits[2]   fruits[3]  fruits[4]
               fruits[-5]  fruits[-4] fruits[-3]  fruits[-2] fruits[-1]
```

- If you `slice` without the second value, it will take everything up to the end of the array

```
fruits.slice(3);   // gives us Apple and Mango
fruits.slice(-1);  // gives us Mango only
```

# Finding something in an array

- If we set up an array…

```
var fruits = ["Banana", "Orange", "Lemon",   "Apple", "Mango"];
             fruits[0]   fruits[1]  fruits[2]   fruits[3]  fruits[4]
```

```
var whereIsTheLemon = fruits.indexOf("Lemon");
```

  - gives us 2…

```
var needThisForBanoffeePie = fruits.indexOf("Banana");
```

  - gives us 0
- so what is indexOf telling us?

# More fun with arrays

- Lots of others…

| Method | Description |
|---|---|
| concat() | Joins two or more arrays, and returns a copy of the joined arrays |
| indexOf() | Search the array for an element and returns it's position |
| join() | Joins all elements of an array into a string |
| lastIndexOf() | Search the array for an element, starting at the end, and returns it's position |
| pop() | Removes the last element of an array, and returns that element |
| push() | Adds new elements to the end of an array, and returns the new length |
| reverse() | Reverses the order of the elements in an array |
| shift() | Removes the first element of an array, and returns that element |
| slice() | Selects a part of an array, and returns the new array |
| sort() | Sorts the elements of an array |
| splice() | Adds/Removes elements from an array |
| toString() | Converts an array to a string, and returns the result |
| unshift() | Adds new elements to the beginning of an array, and returns the new length |
| valueOf() | Returns the primitive value of an array |

- W3Schools is a good reference when getting started:
  - http://www.w3schools.com/jsref/jsref_obj_array.asp

# Slicing and dicing text

- You can use slice on a single string (a text value or variable) rather than an array - it will give you the appropriate bit of the string (known as a *substring*)

```
var name = "Paul Neve";
var surname = name.slice(5,9);              // gives Neve
var surnameAlternative = name.slice(-4); // gives Neve
```

- substr is similar to slice, but the second value changes to be the number of characters to take

```
var surname = name.substr(5,4);             // gives Neve
```

  - **WARNING**: don't use substring instead of substr - confusingly, substring actually exists and is more like slice in its behaviour

# Slicing and dicing text

- You can use charAt to take a single character out of a string:

```
var name = "Paul Neve";
var secondInitial = name.charAt(5);   // gives N
var firstInAlphabet = name.charAt(1); // gives a
```

- You can convert a string to all upper or lower case:

```
var shouting = name.toUpperCase();    // gives PAUL NEVE
var poorGrammar = name.toLowerCase(); // gives paul neve
```

- Many others… again, check out W3Schools
  - http://www.w3schools.com/jsref/jsref_obj_string.asp

# Summary

- The first rule of Javascript is that Java is not Javascript
- **The second rule of Javascript is that Java is NOT Javascript!**
- That said… they are both C-like languages, so share many similarities of grammar and syntax
- Javascript is usually used in conjunction with HTML (web) pages, and any input/output to/from the user has to go through such an HTML page
- In the early exercises, we'll introduce an artificial console that you can print messages to and get typed input from the user on, just so you can get used to the language before having to worry about interactions with HTML

# Summary

- Javascript is **weakly typed** – you don't have to specify what kind of data is stored in your variables when you declare them
- Declare variables with var
- Use the usual C-like base constructs for **`if`**, **`while`**, **`do/while`** and **`for`**
- Arrays in Javascript are great – they can and will resize on the fly
- Lots of functions to slice and dice arrays
- Also lots of functions to slice and dice strings (text)