

JavaScript for ABAP Programmers

Scope

Chris Whealy / The RIG





ABAP

Strongly typed

Syntax similar to COBOL

Block Scope

No equivalent concept

OO using class based inheritance

Imperative programming

JavaScript

Weakly typed

Syntax derived from Java

Lexical Scope

Functions are 1st class citizens

OO using referential inheritance

Imperative or Functional programming



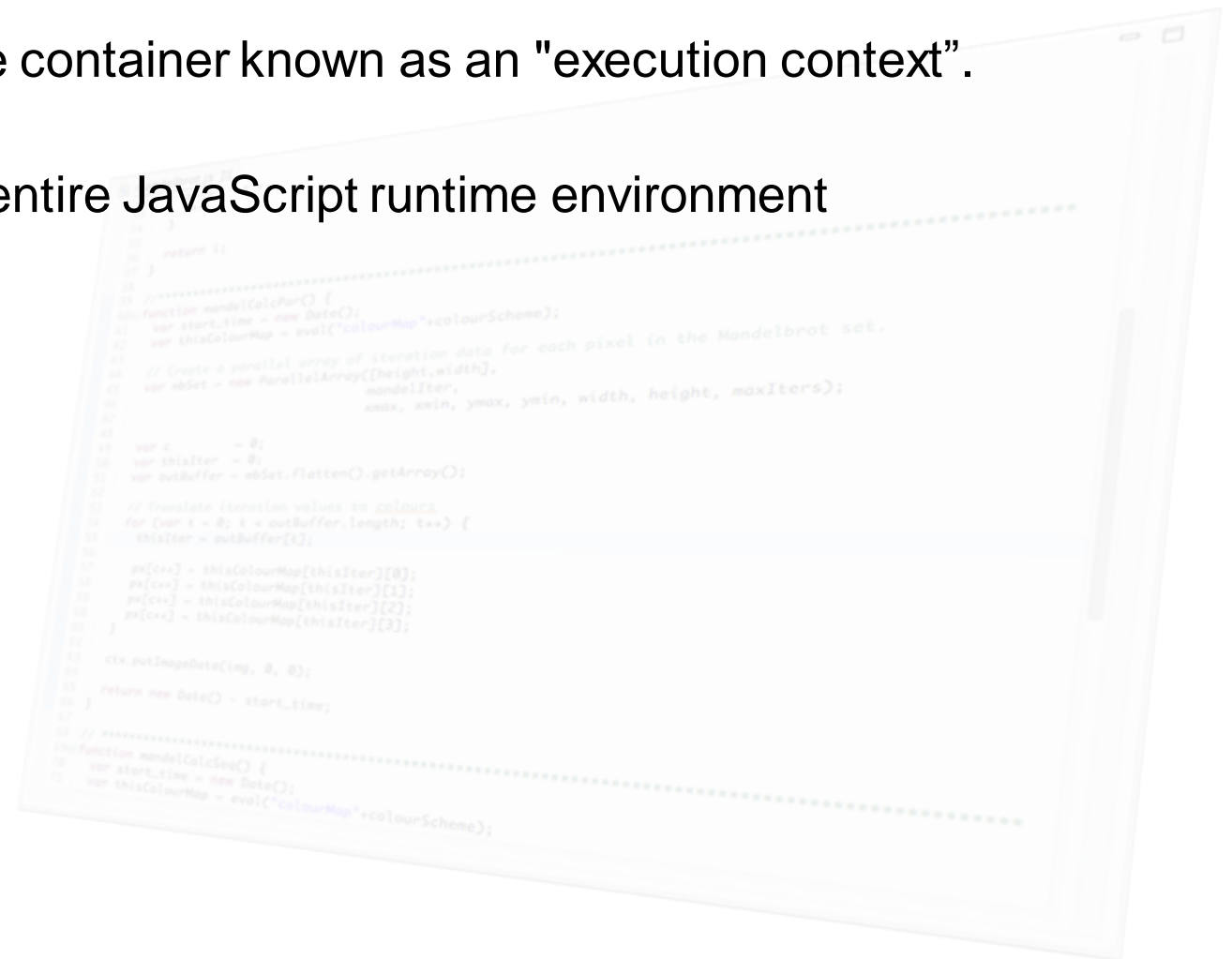
Execution Contexts

Execution Contexts

All JavaScript code is executed within a runtime container known as an "execution context".

There are 3 different execution contexts:

- **The Global Context** : created once for the entire JavaScript runtime environment

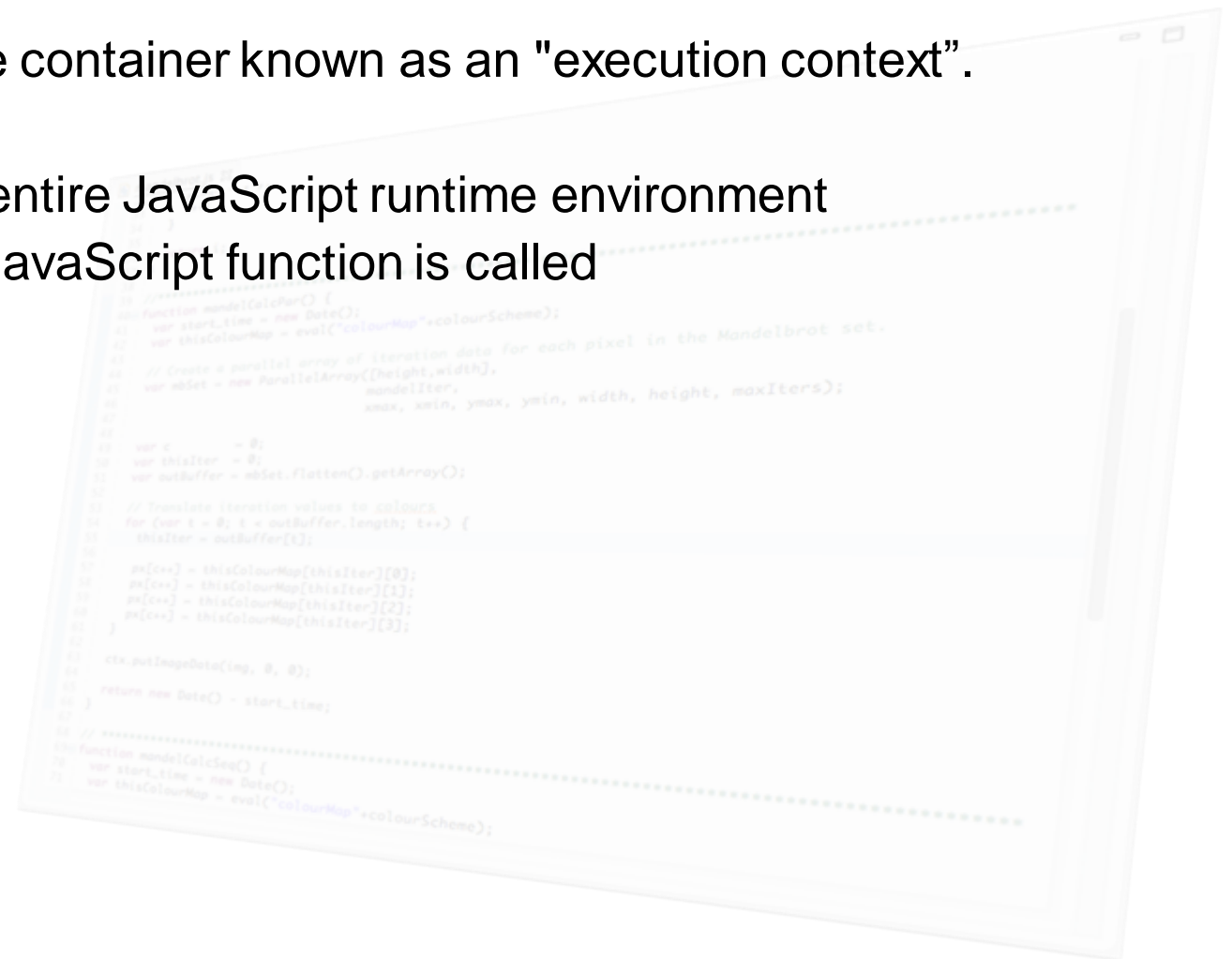


Execution Contexts

All JavaScript code is executed within a runtime container known as an "execution context".

There are 3 different execution contexts:

- **The Global Context** : created once for the entire JavaScript runtime environment
- **A Function Context** : created each time a JavaScript function is called

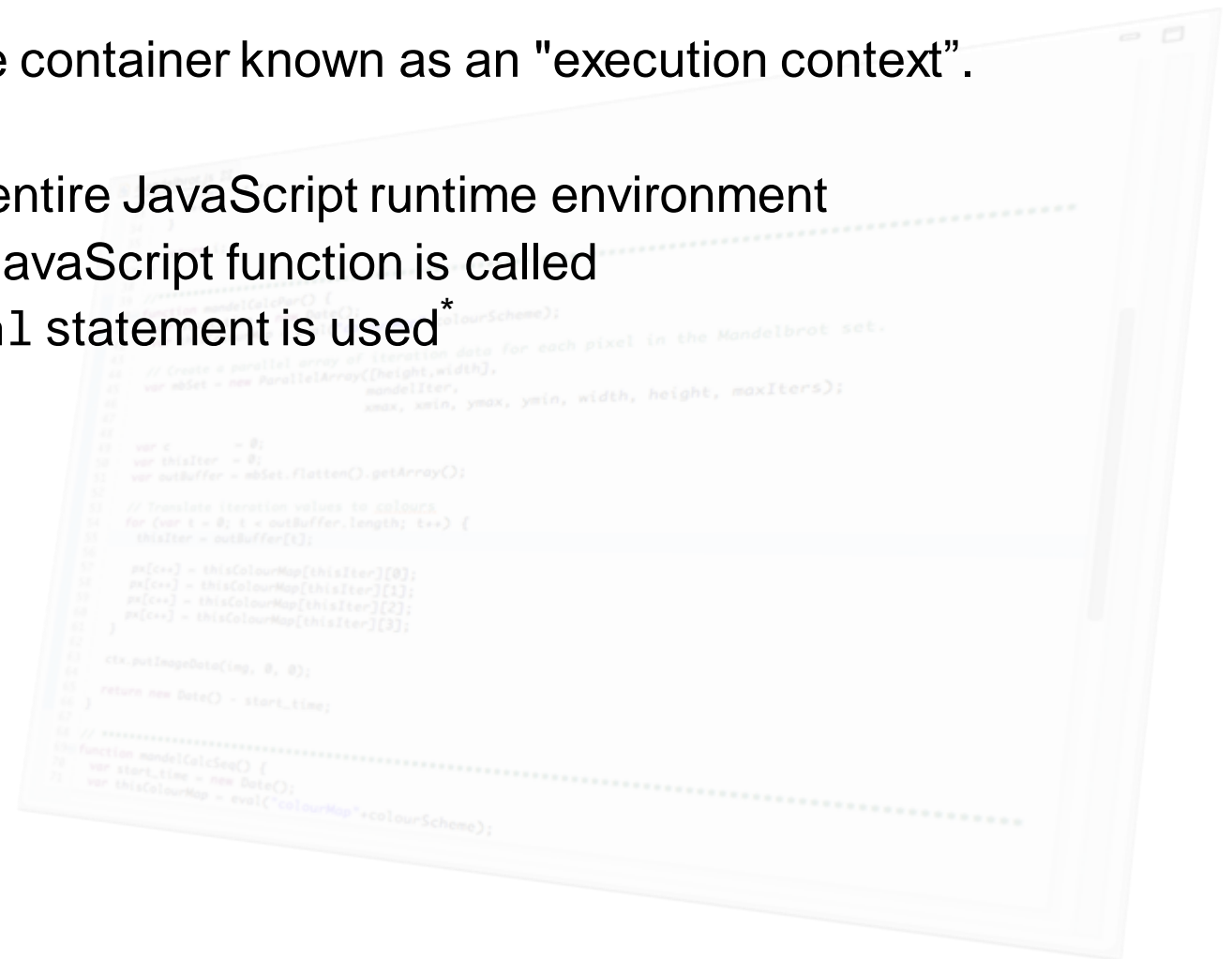


Execution Contexts

All JavaScript code is executed within a runtime container known as an "execution context".

There are 3 different execution contexts:

- **The Global Context** : created once for the entire JavaScript runtime environment
- **A Function Context** : created each time a JavaScript function is called
- **An eval Context** : created when the `eval` statement is used*



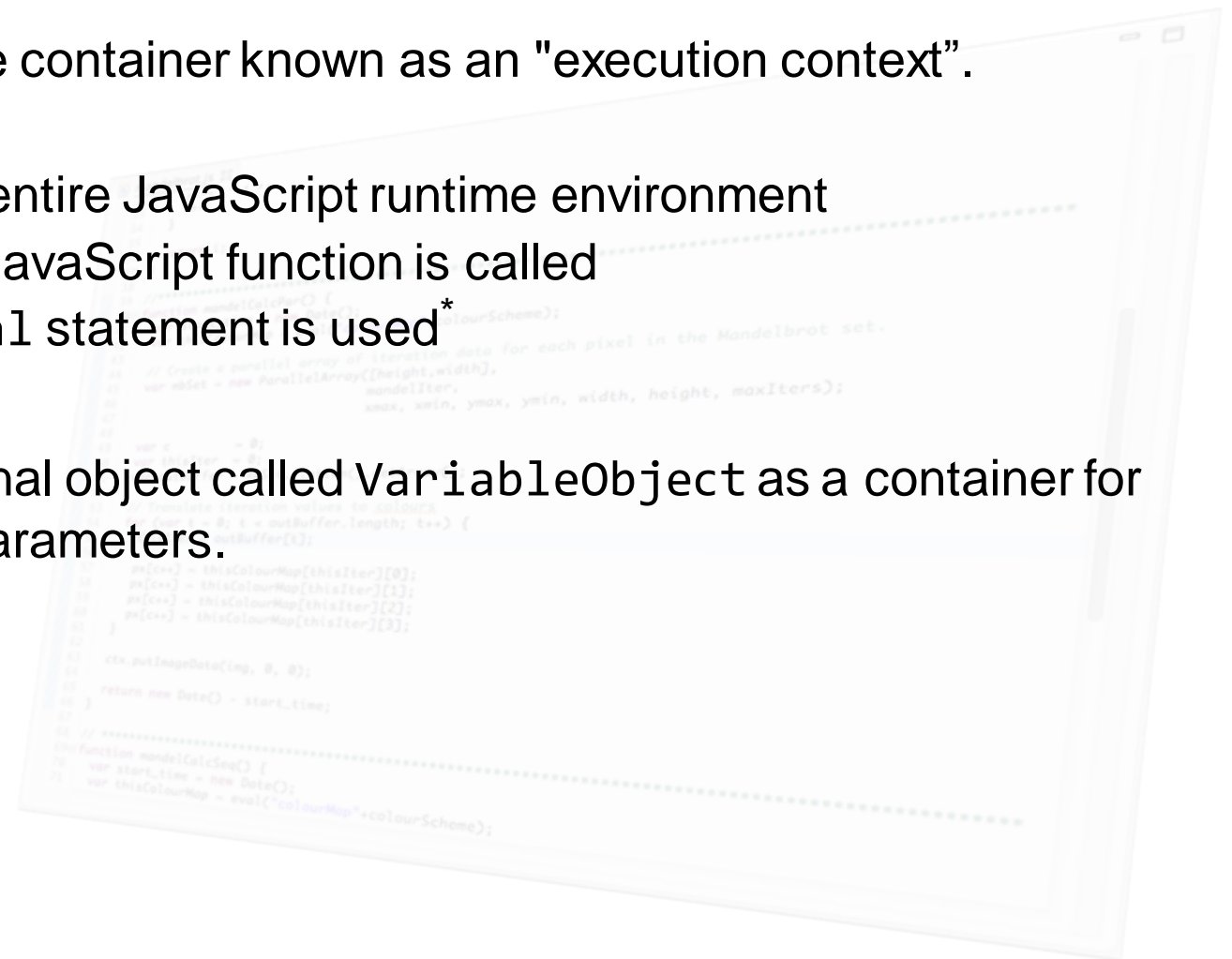
Execution Contexts

All JavaScript code is executed within a runtime container known as an "execution context".

There are 3 different execution contexts:

- **The Global Context** : created once for the entire JavaScript runtime environment
- **A Function Context** : created each time a JavaScript function is called
- **An eval Context** : created when the `eval` statement is used*

Each of these execution contexts uses an internal object called `VariableObject` as a container for storing all its properties, variables and formal parameters.



Execution Contexts

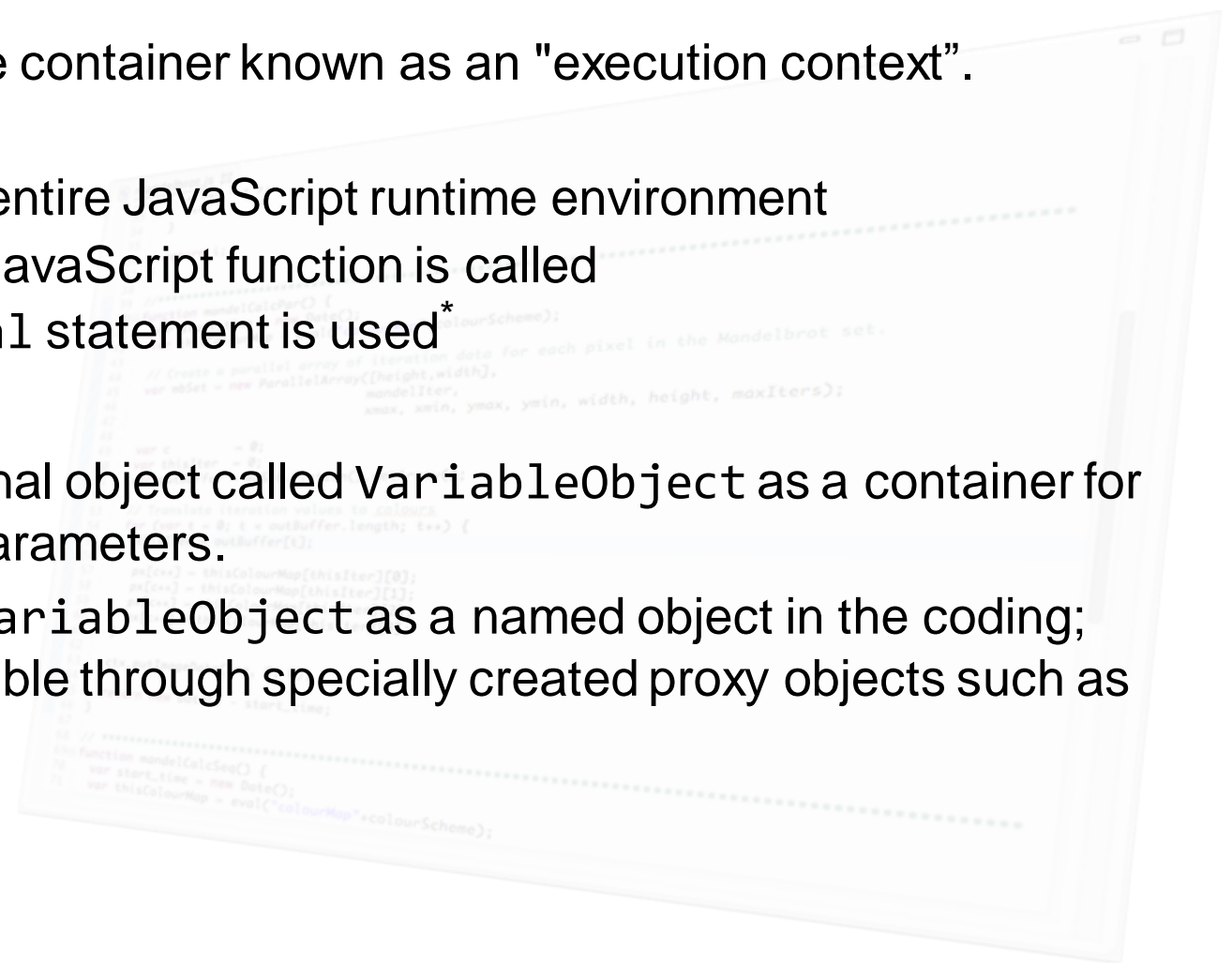
All JavaScript code is executed within a runtime container known as an "execution context".

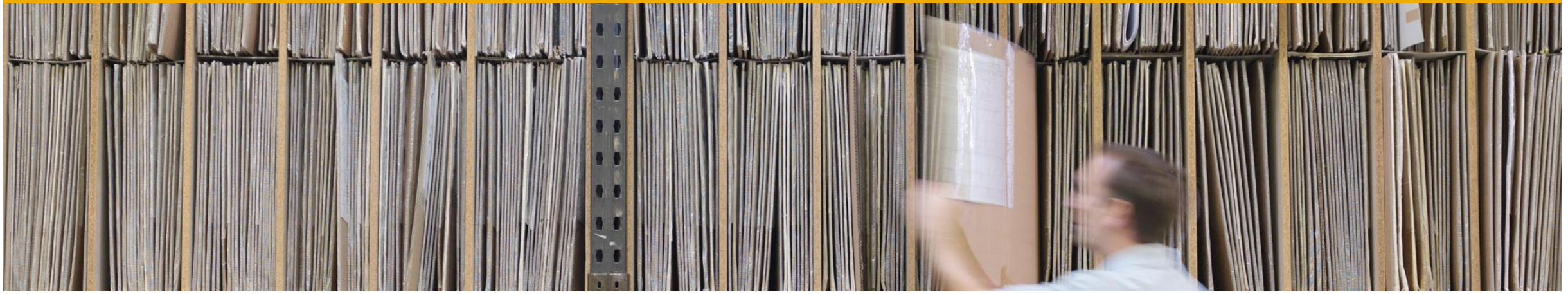
There are 3 different execution contexts:

- **The Global Context** : created once for the entire JavaScript runtime environment
- **A Function Context** : created each time a JavaScript function is called
- **An eval Context** : created when the `eval` statement is used*

Each of these execution contexts uses an internal object called `VariableObject` as a container for storing all its properties, variables and formal parameters.

JavaScript does not give any direct access to `VariableObject` as a named object in the coding; however, the contents of this object are accessible through specially created proxy objects such as `window` or **this**.





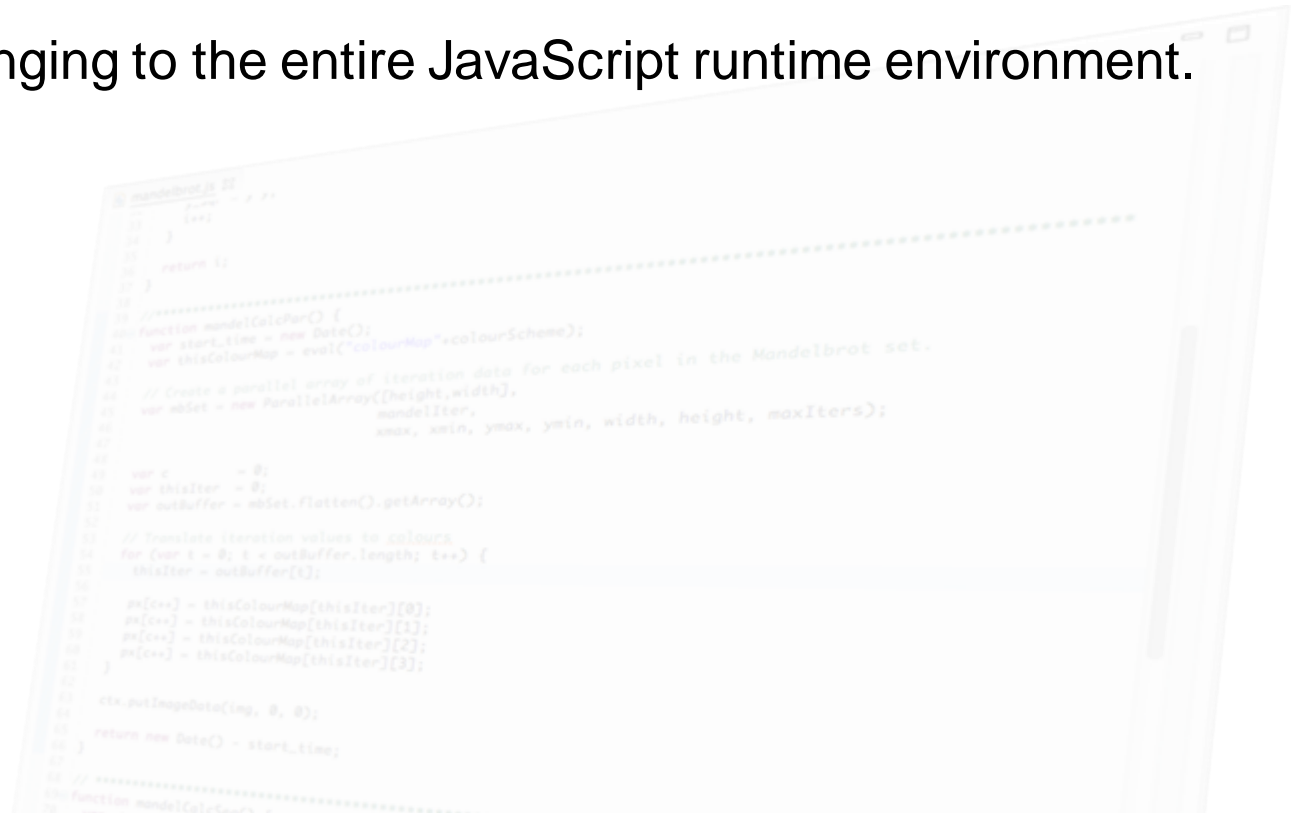
Declarations

The Global Context

Variables, Properties and the difference between them

The Global Context

The global context is the runtime container belonging to the entire JavaScript runtime environment. You will also see the term 'global scope' used.

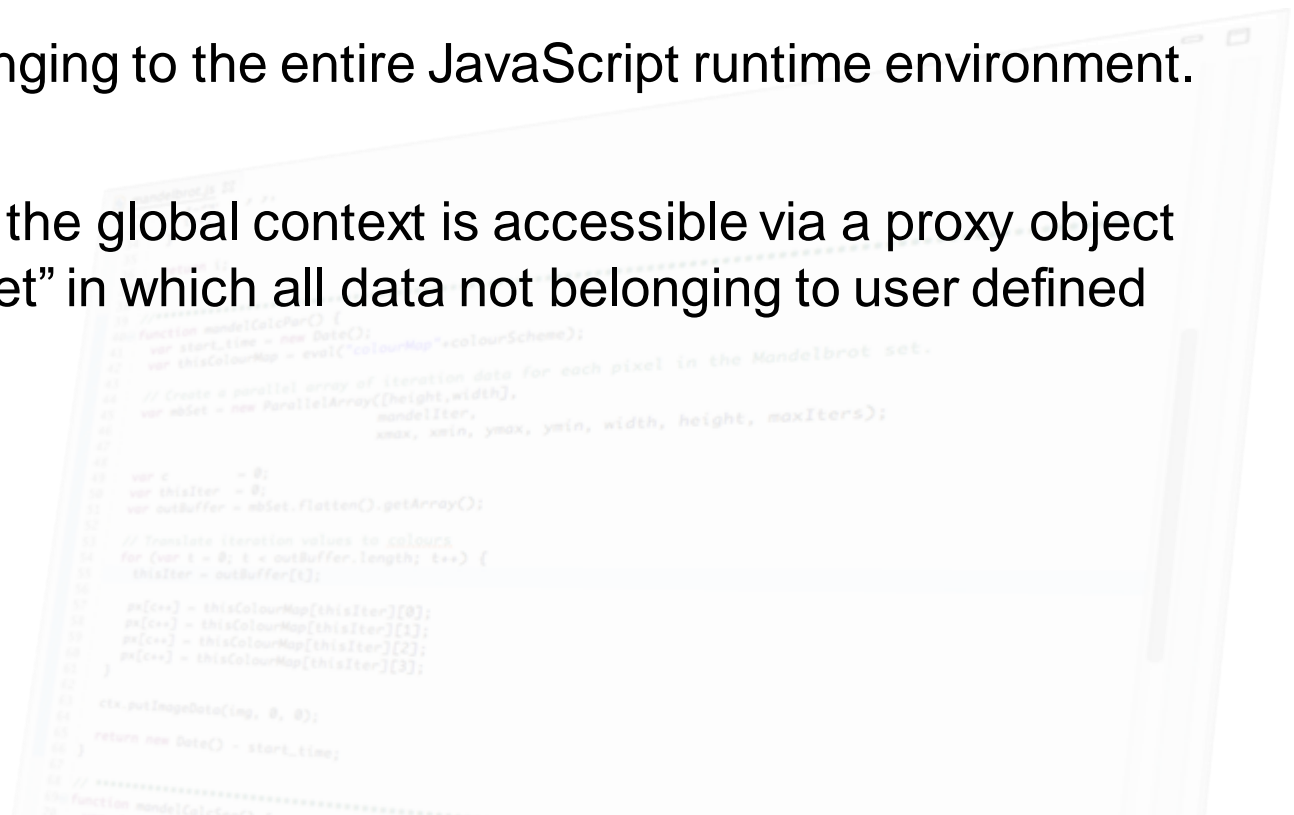


Global Context

The Global Context

The global context is the runtime container belonging to the entire JavaScript runtime environment. You will also see the term ‘global scope’ used.

In a browser, the `VariableObject` belonging to the global context is accessible via a proxy object called `window`. This is a general purpose “bucket” in which all data not belonging to user defined functions is stored.



Global Context: `window`

`document`

`parent`

`top`

`localStorage`

`sessionStorage`

`location`

`history`

`etc...`

Declarations: Variables and Properties in the Global Context 1/2

Since all properties must belong to an object, if a property is created without specifying the object name, then it automatically belongs to the `VariableObject` of the Global Context (I.E. in a browser, this is the window object) and is visible in every execution context.

```
// Create a global property  
foobar = "Phluff 'n' stuff";
```

Global Context: window

foobar

document

parent

top

localStorage

sessionStorage

location

history

etc...

Declarations: Variables and Properties in the Global Context 1/2

Since all properties must belong to an object, if a property is created without specifying the object name, then it automatically belongs to the `VariableObject` of the Global Context (I.E. in a browser, this is the window object) and is visible in every execution context.

```
// Create a global property
foobar = "Phluff 'n' stuff";

foobar;           // → "Phluff 'n' stuff"
window.foobar;    // → "Phluff 'n' stuff"
```

Global properties can be accessed either by direct reference to the property name

Global Context: window

foobar

document

parent

top

localStorage

sessionStorage

location

history

etc...

Declarations: Variables and Properties in the Global Context 1/2

Since all properties must belong to an object, if a property is created without specifying the object name, then it automatically belongs to the `VariableObject` of the Global Context (I.E. in a browser, this is the window object) and is visible in every execution context.

```
// Create a global property
foobar = "Phluff 'n' stuff";

foobar;           // → "Phluff 'n' stuff"
window.foobar;    // → "Phluff 'n' stuff"
```

Or via their parent object and the refinement operator “.”

Global Context: window

foobar

document

parent

top

localStorage

sessionStorage

location

history

etc...

Declarations: Variables and Properties in the Global Context 2/2

The keyword **var** causes a variable to be created. Unfortunately, when a variable is declared in the Global Context, the misleading impression is formed that the variable is no different from any other property of the window object.

```
// Create a global property
foobar = "Phluff 'n' stuff";

foobar;           // → "Phluff 'n' stuff"
window.foobar;    // → "Phluff 'n' stuff"

// Create a global variable
var barfoo = "Chicken soup";
```

Global Context: window

document

parent

top

localStorage

sessionStorage

location

history

etc...

foobar

barfoo

Declarations: Variables and Properties in the Global Context 2/2

The keyword **var** causes a variable to be created. Unfortunately, when a variable is declared in the Global Context, the misleading impression is formed that the variable is no different from any other property of the window object.

```
// Create a global property  
foobar = "Phluff 'n' stuff";
```

```
foobar;           // → "Phluff 'n' stuff"  
window.foobar;    // → "Phluff 'n' stuff"
```

```
// Create a global variable  
var barfoo = "Chicken soup";
```

```
barfoo;           // → "Chicken soup"
```

Global variables are accessed by direct reference to the variable name

Global Context: window

foobar

barfoo

document

parent

top

localStorage

sessionStorage

location

history

etc...

Declarations: Variables and Properties in the Global Context 2/2

The keyword **var** causes a variable to be created. Unfortunately, when a variable is declared in the Global Context, the misleading impression is formed that the variable is no different from any other property of the window object.

```
// Create a global property
foobar = "Phluff 'n' stuff";

foobar;           // → "Phluff 'n' stuff"
window.foobar;    // → "Phluff 'n' stuff"

// Create a global variable
var barfoo = "Chicken soup";

barfoo;           // → "Chicken soup"
window.barfoo;    // → "Chicken soup"
```

This unfortunately makes the variable look just like a property...

Global Context: window

foobar

barfoo

document

parent

top

localStorage

sessionStorage

location

history

etc...

Declarations: Variables and Properties in the Global Context 2/2

The keyword **var** causes a variable to be created. Unfortunately, when a variable is declared in the Global Context, the misleading impression is formed that the variable is no different from any other property of the window object.

```
// Create a global property
foobar = "Phluff 'n' stuff";

foobar;           // → "Phluff 'n' stuff"
window.foobar;    // → "Phluff 'n' stuff"

// Create a global variable
var barfoo = "Chicken soup";

barfoo;           // → "Chicken soup"
window.barfoo;    // → "Chicken soup"
```

A property always belongs to an object and, unless you say otherwise, that object will be the global context.

Global Context: window

foobar

barfoo

document

parent

top

localStorage

sessionStorage

location

history

etc...

Declarations: Variables and Properties in the Global Context 2/2

The keyword `var` causes a variable to be created. Unfortunately, when a variable is declared in the Global Context, the misleading impression is formed that the variable is no different from any other property of the window object.

```
// Create a global property
foobar = "Phluff 'n' stuff";

foobar;           // → "Phluff 'n' stuff"
window.foobar;    // → "Phluff 'n' stuff"

// Create a global variable
var barfoo = "Chicken soup";

barfoo;           // → "Chicken soup"
window.barfoo;    // → "Chicken soup"
```

A property always belongs to an object and, unless you say otherwise, that object will be the global context.

A variable always belongs to an execution context; however, in the global context, the execution context and the global context ***are the same thing***.

Global Context: window

foobar

barfoo

document

parent

top

localStorage

sessionStorage

location

history

etc...

Declarations: Variables and Properties in the Global Context 2/2

The keyword `var` causes a variable to be created. Unfortunately, when a variable is declared in the Global Context, the misleading impression is formed that the variable is no different from any other property of the window object.

```
// Create a global property
foobar = "Phluff 'n' stuff";

foobar;           // → "Phluff 'n' stuff"
window.foobar;    // → "Phluff 'n' stuff"

// Create a global variable
var barfoo = "Chicken soup";

barfoo;           // → "Chicken soup"
window.barfoo;    // → "Chicken soup"
```

A property always belongs to an object and, unless you say otherwise, that object will be the global context.

A variable always belongs to an execution context; however, in the global context, the execution context and the global context **are the same thing**.

Therefore the misleading impression is created that global variables are the same as global properties.

Global Context: window

foobar

barfoo

document

parent

top

localStorage

sessionStorage

location

history

etc...

Declarations: Variables and Properties in the Global Context 2/2

The keyword `var` causes a variable to be created. Unfortunately, when a variable is declared in the Global Context, the misleading impression is formed that the variable is no different from any other property of the window object.

```
// Create a global property
foobar = "Phluff 'n' stuff";

foobar;           // → "Phluff 'n' stuff"
window.foobar;    // → "Phluff 'n' stuff"

// Create a global variable
var barfoo = "Chicken soup";

barfoo;           // → "Chicken soup"
window.barfoo;    // → "Chicken soup"
```

A property always belongs to an object and, unless you say otherwise, that object will be the global context.

A variable always belongs to an execution context; however, in the global context, the execution context and the global context ***are the same thing***.

Therefore the misleading impression is created that global variables are the same as global properties.

Important

This impression exists only for variables in the global context! In function contexts, it does not apply.

Global Context: window

foobar

barfoo

document

parent

top

localStorage

sessionStorage

location

history

etc...



Declarations

Variable Hoisting

Declarations: Variables Are Hoisted!

Understanding the difference between variables and properties is important because it has an impact on the way you should write your code.

A variable is any named value declared using the **var** keyword.

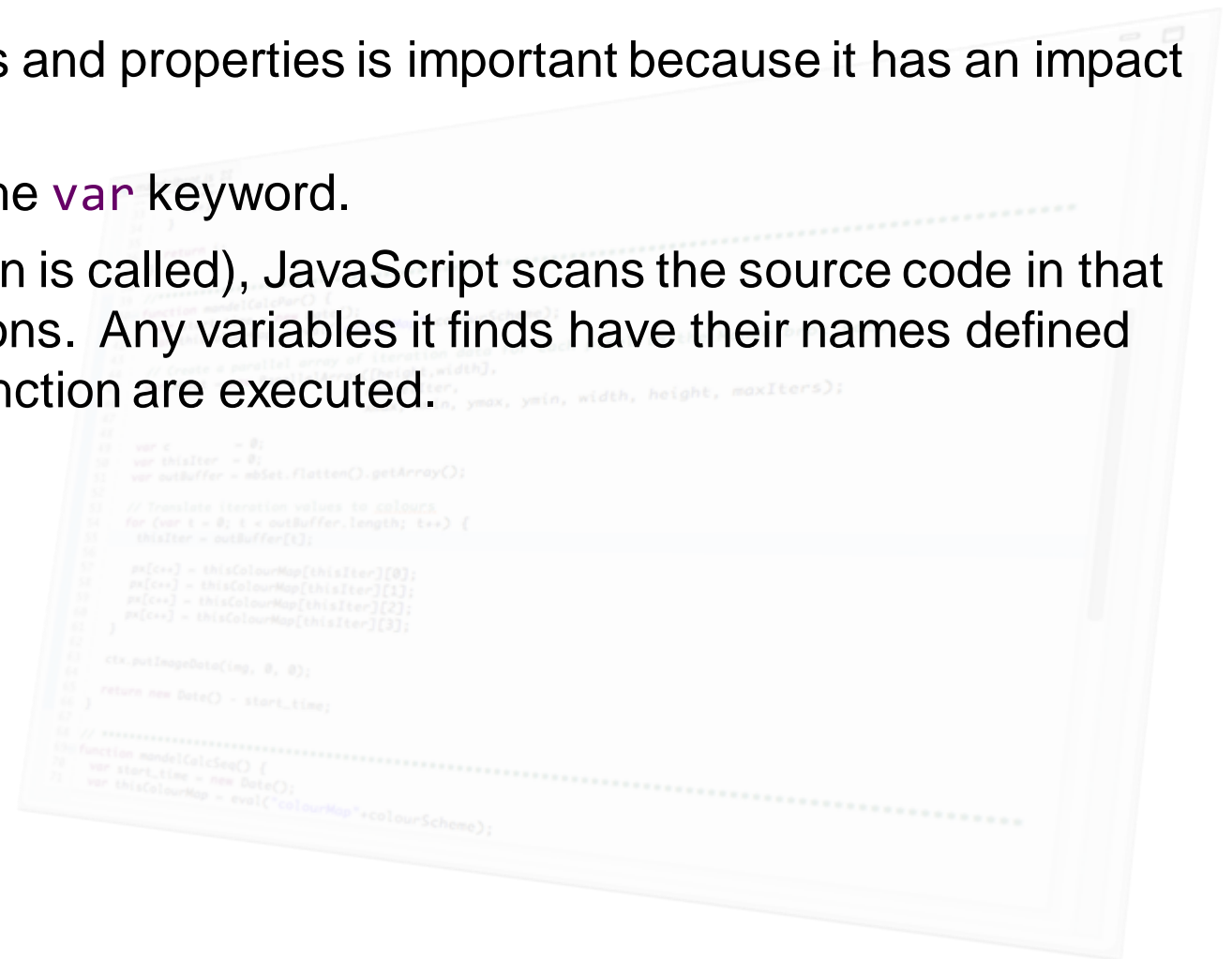


Declarations: Variables Are Hoisted!

Understanding the difference between variables and properties is important because it has an impact on the way you should write your code.

A variable is any named value declared using the **var** keyword.

When an execution context starts (I.E. a function is called), JavaScript scans the source code in that execution context looking for variable declarations. Any variables it finds have their names defined immediately – **before** any statements of that function are executed.



Declarations: Variables Are Hoisted!

Understanding the difference between variables and properties is important because it has an impact on the way you should write your code.

A variable is any named value declared using the `var` keyword.

When an execution context starts (I.E. a function is called), JavaScript scans the source code in that execution context looking for variable declarations. Any variables it finds have their names defined immediately – **before** any statements of that function are executed.

However, the variable will not be assigned a value until execution reaches that statement!



```
12 // Translate iteration values to colours
13 for (var i = 0; i < outBuffer.length; i++) {
14   thisIter = outBuffer[i];
15
16   px[i++] = thisColourMap[thisIter][0];
17   py[i++] = thisColourMap[thisIter][1];
18   pz[i++] = thisColourMap[thisIter][2];
19   pw[i++] = thisColourMap[thisIter][3];
20 }
21
22 ctx.putImageData(img, 0, 0);
23 return new Date() - start_time;
24 }
25
26 // =====
27 function main(calcRes) {
28   var start_time = new Date();
29   var thisColourMap = eval("colourMap" + colourScheme);
```

Declarations: Variables Are Hoisted!

Understanding the difference between variables and properties is important because it has an impact on the way you should write your code.

A variable is any named value declared using the `var` keyword.

When an execution context starts (I.E. a function is called), JavaScript scans the source code in that execution context looking for variable declarations. Any variables it finds have their names defined immediately – **before** any statements of that function are executed.

However, the variable will not be assigned a value until execution reaches that statement!

```
// Doing things backwards, but getting away with it (kinda)
alert(someNumber);           // → undefined. Because of hoisting the variable name is known, but has no value yet

var someNumber = 10;         // Now assign it a value

alert(someNumber);           // → 10
```

Declarations: Variables Are Hoisted!

Understanding the difference between variables and properties is important because it has an impact on the way you should write your code.

A variable is any named value declared using the `var` keyword.

When an execution context starts (I.E. a function is called), JavaScript scans the source code in that execution context looking for variable declarations. Any variables it finds have their names defined immediately – **before** any statements of that function are executed.

However, the variable will not be assigned a value until execution reaches that statement!

```
// Doing things backwards, but getting away with it (kinda)
alert(someNumber);           // → undefined. Because of hoisting the variable name is known, but has no value yet

var someNumber = 10;         // Now assign it a value

alert(someNumber);           // → 10
```

This situation is known as "hoisting" because JavaScript behaves as if all variable declarations are raised (or hoisted) to the top of the function's source code.

Declarations: Properties Are Created Sequentially!

A property is created simply by assigning a value to a name. However, unlike variable name declarations, properties are not hoisted. Their creation takes place in a strictly sequentially manner.

The property name will remain unknown to JavaScript until execution reaches the statement.

Depending on how the property is referenced, this might result in a runtime reference error.

```
// Doing things backwards and not always getting away with it
alert(window.someProperty); // → undefined. The window object is known, but the property name isn't
```

Declarations: Properties Are Created Sequentially!

A property is created simply by assigning a value to a name. However, unlike variable name declarations, properties are not hoisted. Their creation takes place in a strictly sequentially manner.

The property name will remain unknown to JavaScript until execution reaches the statement.

Depending on how the property is referenced, this might result in a runtime reference error.

```
// Doing things backwards and not always getting away with it
alert(window.someProperty); // → undefined. The window object is known, but the property name isn't
alert(someProperty);        // → KABOOM! (Reference error) The unqualified property name is unknown!
```


Declarations: Properties Are Created Sequentially!

A property is created simply by assigning a value to a name. However, unlike variable name declarations, properties are not hoisted. Their creation takes place in a strictly sequentially manner.

The property name will remain unknown to JavaScript until execution reaches the statement.

Depending on how the property is referenced, this might result in a runtime reference error.

```
// Doing things backwards and not always getting away with it
alert(window.someProperty); // → undefined. The window object is known, but the property name isn't
alert(someProperty);        // → KABOOM! (Reference error) The unqualified property name is unknown!

someProperty = 10;          // This statement both creates the global property and assigns it a value
```

Declarations: Properties Are Created Sequentially!

A property is created simply by assigning a value to a name. However, unlike variable name declarations, properties are not hoisted. Their creation takes place in a strictly sequentially manner.

The property name will remain unknown to JavaScript until execution reaches the statement.

Depending on how the property is referenced, this might result in a runtime reference error.

```
// Doing things backwards and not always getting away with it
alert(window.someProperty); // → undefined. The window object is known, but the property name isn't
alert(someProperty);        // → KABOOM! (Reference error) The unqualified property name is unknown!

someProperty = 10;          // This statement both creates the global property and assigns it a value

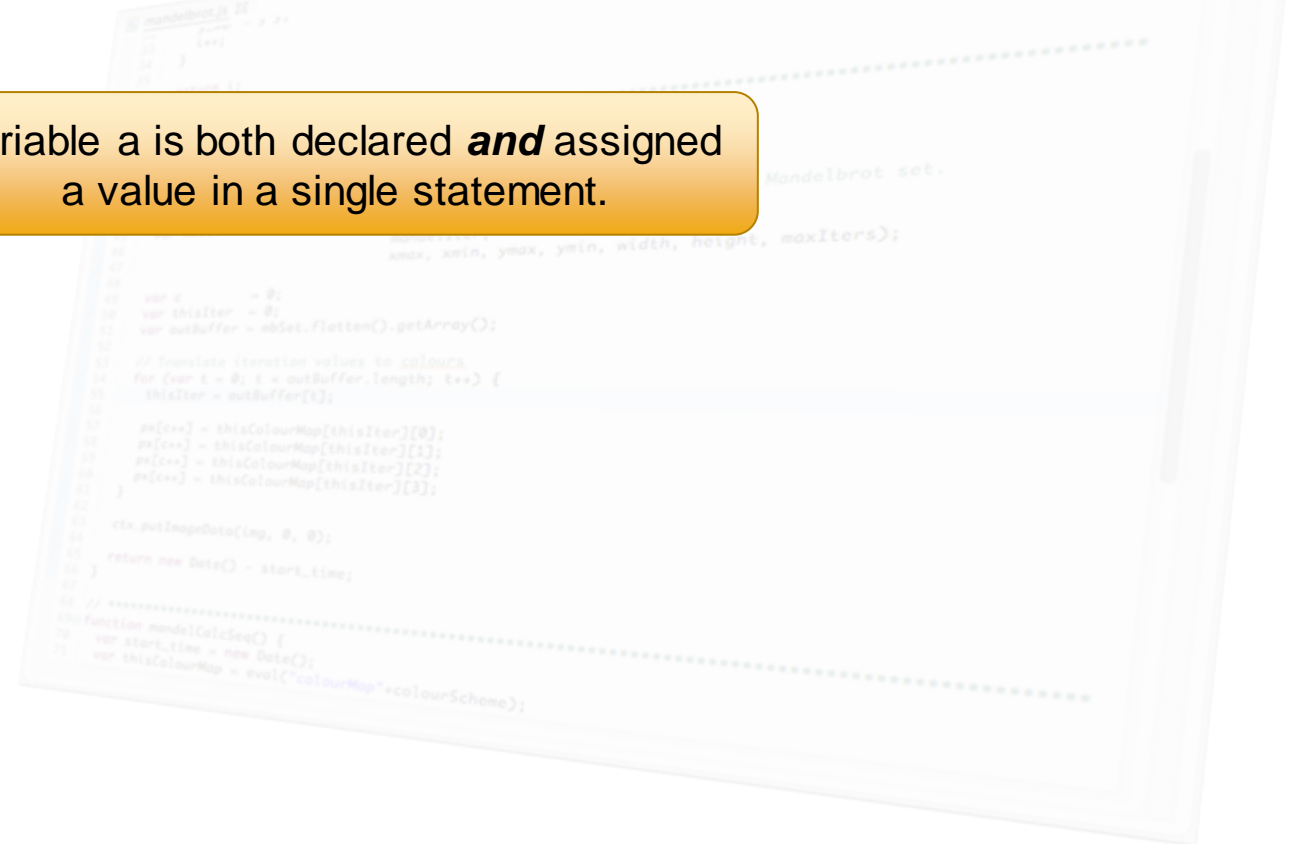
alert(someProperty);        // → 10
```

Declarations: Consequences of Hoisting 1/2

Due to the fact that all JavaScript variables are hoisted, the location of a variable's **declaration** within the source code of a function is not important.

```
function test() {  
  var a = 10;  
  
  for (var i=0; i<5; i++) {  
    console.log("a + i = " + (a + i));  
  }  
}
```

Variable a is both declared **and** assigned a value in a single statement.



Declarations: Consequences of Hoisting 1/2

Due to the fact that all JavaScript variables are hoisted, the location of a variable's **declaration** within the source code of a function is not important.

```
function test() {  
  var a = 10;  
  
  for (var i=0; i<5; i++) {  
    console.log("a + i = " + (a + i));  
  }  
}
```

This gives the expected runtime results

a + i = 10
a + i = 11
a + i = 12
a + i = 13
a + i = 14

Declarations: Consequences of Hoisting 2/2

However due to the effects of hoisting, the location of the first **assignment** to a variable could have serious consequences for the logic of your coding!

```
function test() {  
  for (var i=0; i<5; i++) {  
    console.log("a + i = " + (a + i));  
  }  
  
  var a = 10;  
}
```

Moving the declaration to the end of the function has no impact on whether JavaScript “knows” the name of this variable...

Declarations: Consequences of Hoisting 2/2

However due to the effects of hoisting, the location of the first **assignment** to a variable could have serious consequences for the logic of your coding!

```
function test() {  
  for (var i=0; i<5; i++) {  
    console.log("a + i = " + (a + i));  
  }  
  
  var a = 10;  
}
```

However, the assignment of the variable's value has also been moved.

The logic of the coding is now broken!

<code>a + i = NaN</code>
<code>a + i = NaN</code>
<code>a + i = NaN</code>
<code>a + i = NaN</code>
<code>a + i = NaN</code>

Declarations: Consequences of Hoisting 2/2

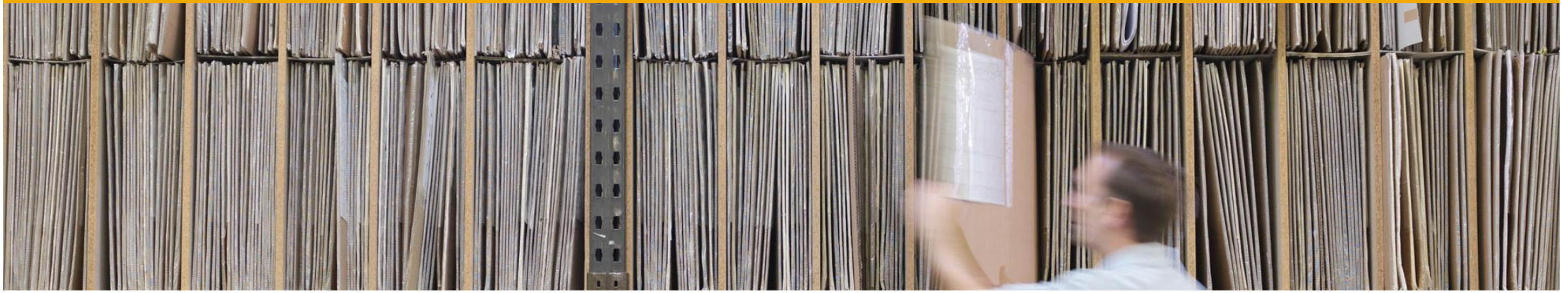
However due to the effects of hoisting, the location of the first **assignment** to a variable could have serious consequences for the logic of your coding!

```
function test() {  
  for (var i=0; i<5; i++) {  
    console.log("a + i = " + (a + i));  
  }  
  
  var a = 10;  
}
```

IMPORTANT!

Even though JavaScript knows about all the declared variables in a function **before** the first statement of the function is executed, the value of those variables will always be **undefined** until the flow of execution reaches a statement that assigns the variable a value!

a + i = NaN
a + i = NaN
a + i = NaN
a + i = NaN
a + i = NaN



Declarations

Named and Anonymous Functions
Automatic Function Execution

Declarations: Creating Named and Anonymous Functions

A function can be created using syntax similar to that seen in other programming languages such as Java and C. Here, a function object is created that has the explicit name of person.

```
// Create a named function
function person() {
    var firstName = "Harry";
    var lastName  = "Hawk";
    var hobbies   = ["swimming ", "cycling"];

    var getHobby = function(n) {
        return firstName + " likes " + hobbies[n];
    }
}
```



Declarations: Creating Named and Anonymous Functions

A function can be created using syntax similar to that seen in other programming languages such as Java and C. Here, a function object is created that has the explicit name of person.

However, there is a subtle variation that allows a more generic approach to be taken. This is where an anonymous function object is created. This object is then stored as the value of a variable.

```
// Create a named function
function person() {
    var firstName = "Harry";
    var lastName  = "Hawk";
    var hobbies   = ["swimming ", "cycling"];

    var getHobby = function(n) {
        return firstName + " likes " + hobbies[n];
    }
}
```

```
// Create an anonymous function and assign it to
// a variable
var person = function() {
    var firstName = "Harry";
    var lastName  = "Hawk";
    var hobbies   = ["swimming ", "cycling"];

    var getHobby = function(n) {
        return firstName + " likes " + hobbies[n];
    }
}
```


Declarations: Creating Named and Anonymous Functions

A function can be created using syntax similar to that seen in other programming languages such as Java and C. Here, a function object is created that has the explicit name of person.

However, there is a subtle variation that allows a more generic approach to be taken. This is where an anonymous function object is created. This object is then stored as the value of a variable.

```
// Create a named function
function person() {
    var firstName = "Harry";
    var lastName  = "Hawk";
    var hobbies   = ["swimming ", "cycling"];

    var getHobby = function(n) {
        return firstName + " likes " + hobbies[n];
    }
}
```

```
// Create an anonymous function and assign it to
// a variable
var person = function() {
    var firstName = "Harry";
    var lastName  = "Hawk";
    var hobbies   = ["swimming ", "cycling"];

    var getHobby = function(n) {
        return firstName + " likes " + hobbies[n];
    }
}
```

IMPORTANT

A JavaScript function is an object like any other object that additionally has “an executable part”.

Automatic Execution of Functions 1/2

As a consequence of a JavaScript function being nothing more than an object that happens to have an executable part, we need some means of distinguishing between the “***function as data***” and the “***function as an executable unit of code***”. This is where the invocation operator () is used.

Here we see that the function is first defined

```
// Set variable 'person' equal to an anonymous function
var person = function(fName, lName, hobbies) {
    var firstName = fName || "";
    var lastName  = lName || "";
    var hobbies   = hobbies || [];

    return {
        firstName: this.firstName,
        lastName  : this.lastName,
        hobbies   : this.getHobby
    }
}
```

Automatic Execution of Functions 1/3

As a consequence of a JavaScript function being nothing more than an object that happens to have an executable part, we need some means of distinguishing between the “***function as data***” and the “***function as an executable unit of code***”. This is where the invocation operator () is used.

Here we see that the function is first defined, then in a separate statement, is invoked.

```
// Set variable 'person' equal to an anonymous function
var person = function(fName, lName, hobbies) {
    var firstName = fName || "";
    var lastName  = lName || "";
    var hobbies   = hobbies || [];

    return {
        firstName: this.firstName,
        lastName  : this.lastName,
        hobbies   : this.getHobby
    }
}

// Invoke function person to obtain the required object
var someGuy = person("Harry", "Hawk", ["swimming ", "cycling"]);
```

Automatic Execution of Functions 2/3

The previous approach is inconvenient in that we might not care about keeping a reference to the function that generated the person object. Also, the function must be declared and invoked in separate statements. It would be much more convenient if we could have a direct assignment of the required object by invoking the function automatically.

```
// Set variable 'person' equal to an anonymous function
```

```
var someGuy = (function(fName,lName,hobbies) {
```

```
    var firstName = fName || "";
```

```
    var lastName = lName || "";
```

```
    var hobbies = hobbies || [];
```

```
    return {
```

```
        firstName: this.firstName,
```

```
        lastName : this.lastName,
```

```
        hobbies  : this.getHobby
```

```
    }
```

```
})();
```

First, the entire function definition is placed within parentheses. Here, parentheses are being used as the “grouping operator”

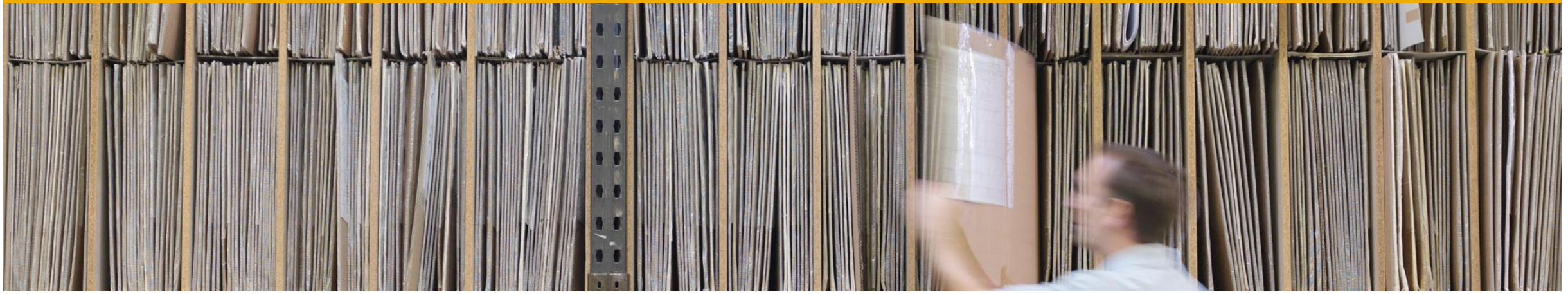
Automatic Execution of Functions 3/3

The previous approach is inconvenient in that we might not care about keeping a reference to the function that generated the person object. Also, the function must be declared and invoked in separate statements. It would be much more convenient if we could have a direct assignment of the required object by invoking the function automatically.

```
// Set variable 'person' equal to an anonymous function
var someGuy = (function(fName,lName,hobbies) {
    var firstName = fName || "";
    var lastName  = lName || "";
    var hobbies   = hobbies || [];

    return {
        firstName: this.firstName,
        lastName  : this.lastName,
        hobbies   : this.getHobby
    }
})("Harry", "Hawk", ["swimming ", "cycling"])
```

Second, the anonymous function inside the parentheses is invoked immediately using the invocation operator plus the relevant parameters



Declarations

Function Context and Lexical Scope

Function Context 1/2

Each time a JavaScript function is called, a new function context is created. This is a private runtime container that has its own `VariableObject`.

```
function() {  
  
}
```

Global Context: window

Function Context:

document

parent

top

localStorage

sessionStorage

location

history

etc...

Function Context 1/2

Each time a JavaScript function is called, a new function context is created. This is a private runtime container that has its own `VariableObject`.

The key difference from the global context is that there is no object equivalent to `window` that would give you access to the function context's `VariableObject`.

```
function() {  
  var localVar = "local";  
}
```

Global Context: `window`

Function Context:

`localVar`

`document`

`parent`

`top`

`localStorage`

`sessionStorage`

`location`

`history`

`etc...`

Function Context 1/2

Each time a JavaScript function is called, a new function context is created. This is a private runtime container that has its own `VariableObject`.

The key difference from the global context is that there is no object equivalent to `window` that would give you access to the function context's `VariableObject`.

This means that you **cannot** create function properties using the same syntax as for global properties.

```
function() {  
  var localVar = "local";  
}
```

Global Context: `window`

Function Context:

`localVar`

`document`

`parent`

`top`

`localStorage`

`sessionStorage`

`location`

`history`

`etc...`

Function Context 1/2

Each time a JavaScript function is called, a new function context is created. This is a private runtime container that has its own `VariableObject`.

The key difference from the global context is that there is no object equivalent to `window` that would give you access to the function context's `VariableObject`.

This means that you **cannot** create function properties using the same syntax as for global properties.

```
function() {  
  var localVar = "local";  
  globalProp = "global";  
}
```

If you try to create a local property using the syntax seen here, you will end up creating a global property!

Global Context: `window`

Function Context:

`localVar`

`globalProp`

`document`

`parent`

`top`

`localStorage`

`sessionStorage`

`location`

`history`

`etc...`

Function Context 2/2

If you wish to create a property that belongs to a function object, then the assignment must identify the local execution context via the special object **this**.

```
function() {  
  var localVar = "local";  
  this.localProp = "local";  
}
```

Global Context: window

Function Context:

localVar

localProp

document

parent

top

localStorage

sessionStorage

location

history

etc...

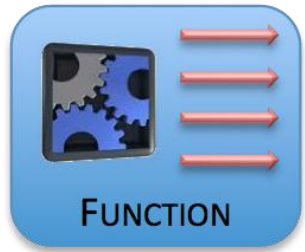
Variable and Object Property Visibility

JavaScript does not directly allow you to specify the visibility of an object's properties. In other words, there is no direct concept of a **public** or **private** object property.



Variable and Object Property Visibility

JavaScript does not directly allow you to specify the visibility of an object's properties. In other words, there is no direct concept of a **public** or **private** object property.



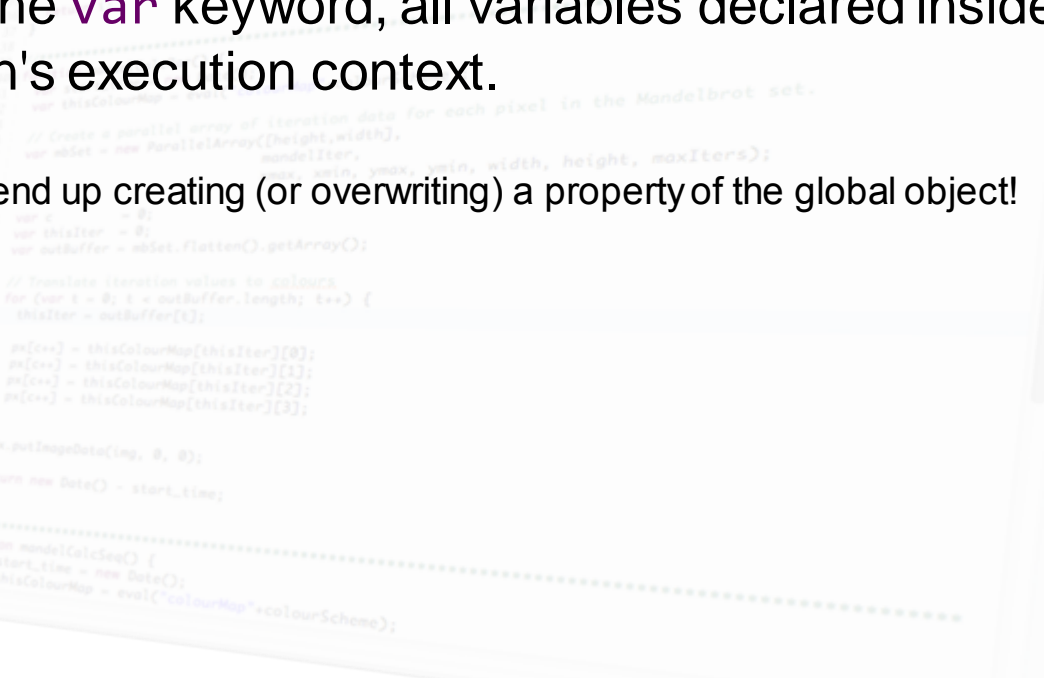
As long as you remember to use the **var** keyword, all variables declared inside a function are private to that function's execution context.

Remember

If you forget to use the **var** keyword, then you'll end up creating (or overwriting) a property of the global object!

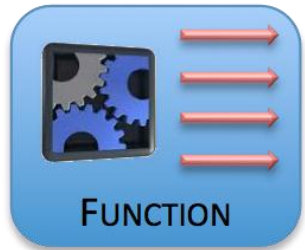
use the **var** keyword, all variables declared inside a function's execution context.

When you'll end up creating (or overwriting) a property of the global object!



Variable and Object Property Visibility

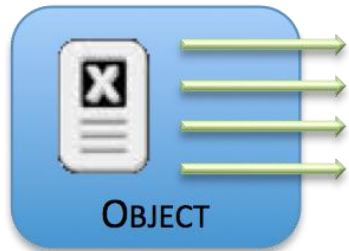
JavaScript does not directly allow you to specify the visibility of an object's properties. In other words, there is no direct concept of a **public** or **private** object property.



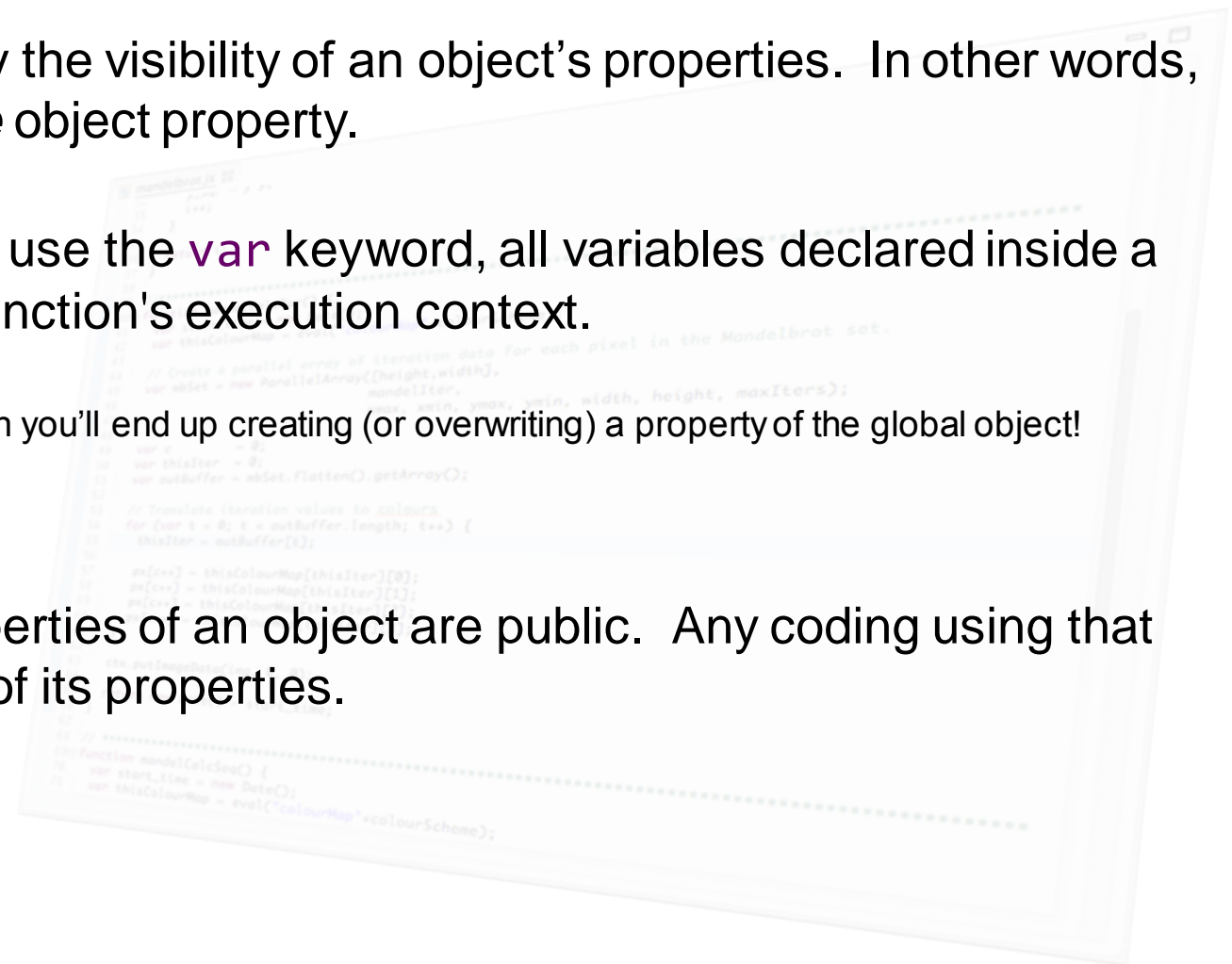
As long as you remember to use the **var** keyword, all variables declared inside a function are private to that function's execution context.

Remember

If you forget to use the **var** keyword, then you'll end up creating (or overwriting) a property of the global object!

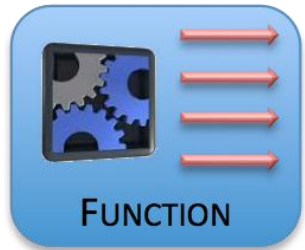


All variables defined as properties of an object are public. Any coding using that object has full access to **all** of its properties.



Variable and Object Property Visibility

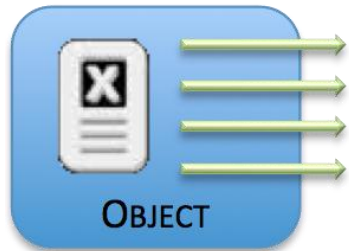
JavaScript does not directly allow you to specify the visibility of an object's properties. In other words, there is no direct concept of a **public** or **private** object property.



As long as you remember to use the **var** keyword, all variables declared inside a function are private to that function's execution context.

Remember

If you forget to use the **var** keyword, then you'll end up creating (or overwriting) a property of the global object!



All variables defined as properties of an object are public. Any coding using that object has full access to **all** of its properties.

This fact will have important consequences for how we code JavaScript objects – but more about this later.

Lexical Scope 1/2

The 'lexical scope' of an object or variable is statically defined by the physical placement of the declaration within the source code. Nested function declarations therefore create scope hierarchies.

```
function outer() {
```

Global Context: window

Function Context: outer

document

parent

top

localStorage

sessionStorage

location

history

etc...

Lexical Scope 1/2

The 'lexical scope' of an object or variable is statically defined by the physical placement of the declaration within the source code. Nested function declarations therefore create scope hierarchies.

```
function outer() {  
  function inner() {  
  
  }  
  return inner();  
}
```

Global Context: window

Function Context: outer

Function Context: inner

document

parent

top

localStorage

sessionStorage

location

history

etc...

Lexical Scope 1/2

The 'lexical scope' of an object or variable is statically defined by the physical placement of the declaration within the source code. Nested function declarations therefore create scope hierarchies.

```
function outer() {  
  function inner() {  
    var sillyJoke1 = 'I've just started a band called "1023Mb";  
    var sillyJoke2 = 'We don't have a gig yet';  
    return sillyJoke1 + "\n" + sillyJoke2;  
  }  
  return inner();  
}
```

Global Context: window

Function Context: outer

Function Context: inner

sillyJoke1

sillyJoke2

document

parent

top

localStorage

sessionStorage

location

history

etc...

Lexical Scope 2/2

Any variable declared using the **var** keyword belongs to the lexical scope of the execution context within which it was declared, and is **private** to that execution context!

```
function giggle() {  
  var sillyJoke1 = 'I\'ve just started a band called "1023Mb";  
  var sillyJoke2 = 'We don\'t have a gig yet';  
  
  return sillyJoke1 + "\n" + sillyJoke2;  
}
```

sillyJoke1; // → Undefined. This variable is not visible to the global execution context

Global Context: window

Function Context: outer

Function Context: inner

sillyJoke1

sillyJoke2

document

parent

top

localStorage

sessionStorage

location

history

etc...

Lexical Scope: Understanding the Meaning of this 1/3

Each time a JavaScript function is invoked, a new function context is created within the scope of the current execution context. This is the runtime container within which (among other things) all the variables used by that function exist.

A special object called **this** gives access to all the variables in your current execution context.

```
// Direct reference to 'this' from code running within the global scope  
this;    // → window
```

In the above example, a direct reference is made to **this** from coding outside the scope of a user created function; therefore, **this** refers to JavaScript's global scope (in a browser, this will be the window object).

Lexical Scope: Understanding the Meaning of this 2/3

Remember that JavaScript functions are simply objects that “have an executable part”.

The value of **this** always points to a function’s execution context; however, the exact identity of the execution context ***varies according to how the function is invoked!***

```
function myGlobalFunc() {  
    return this;  
}  
  
myGlobalFunc(); // → window
```

Lexical Scope: Understanding the Meaning of this 2/3

Remember that JavaScript functions are simply objects that “have an executable part”.

The value of **this** always points to a function’s execution context; however, the exact identity of the execution context **varies according to how the function is invoked!**

```
function myGlobalFunc() {  
  return this;  
}  
  
myGlobalFunc(); // → window
```

Similar to the previous example, the function `myGlobalFunc` is invoked by coding within the global context (I.E. outside the scope of some other function), therefore **this** resolves to the object that identifies the Global Context.

When JavaScript code is run within a browser, the Global Context is identified by the window object. Calling a function from the global context is known as a **baseless function call**.

Lexical Scope: Understanding the Meaning of this 3/3

When the property of an object is defined as a function, then that function is known as a **method**. In this example, the `whatThis` property is a function (or method) belonging to the object `myObj`. The parent object `myObj` can now act as the execution context for function `whatThis`.

```
var myObj = {  
  whatThis : function() {  
    return this;  
  }  
}  
  
// Two different ways of calling method whatThis(). Both refer to the method via object myObj  
myObj.whatThis();           // → myObj - because myObj is the execution context of function whatThis  
myObj['whatThis']();        // → myObj
```

As long as method `whatThis()` is called via the object `myObj`, then **this** will resolve to `myObj`.

Lexical Scope: Flexibility in the Meaning of this

However, it is perfectly possible to call a method using some other object as the execution context. This gives you great flexibility – especially in the area of code reuse. However, you must be very clear in your mind to ensure that the meaning of **this** resolves to the intended object!

For example:

```
var myObj = {  
  whatsThis : function() {  
    return this;  
  }  
}
```

Lexical Scope: Flexibility in the Meaning of this

However, it is perfectly possible to call a method using some other object as the execution context. This gives you great flexibility – especially in the area of code reuse. However, you must be very clear in your mind to ensure that the meaning of **this** resolves to the intended object!

For example:

```
var myObj = {  
  whatsThis : function() {  
    return this;  
  }  
}  
  
// Create some other object that contains a method that refers to the whatsThis method of object myObj  
var someOtherObject = {};  
someOtherObject.anotherFunction = myObj.whatsThis;  
someOtherObject.anotherFunction();    // → someOtherObject.    Called from the execution context of someOtherObject
```

Lexical Scope: Flexibility in the Meaning of this

However, it is perfectly possible to call a method using some other object as the execution context. This gives you great flexibility – especially in the area of code reuse. However, you must be very clear in your mind to ensure that the meaning of **this** resolves to the intended object!

For example:

```
var myObj = {  
  whatsThis : function() {  
    return this;  
  }  
}  
  
// Create some other object that contains a method that refers to the whatsThis method of object myObj  
var someOtherObject = {};  
someOtherObject.anotherFunction = myObj.whatsThis;  
someOtherObject.anotherFunction();    // → someOtherObject. Called from the execution context of someOtherObject  
  
// Create a global variable whose value is a reference to the whatsThis method of object myObj  
var aGlobalVariable = myObj.whatsThis;  
aGlobalVariable();                    // → window. Called from the global execution context
```

Lexical Scope: Flexibility in the Meaning of this

However, it is perfectly possible to call a method using some other object as the execution context. This gives you great flexibility – especially in the area of code reuse. However, you must be very clear in your mind to ensure that the meaning of **this** resolves to the intended object!

For example:

```
var myObj = {  
  whatsThis : function() {  
    return this;  
  }  
}  
  
// Create some other object that contains a method that refers to the whatsThis method of object myObj  
var someOtherObject = {};  
someOtherObject.anotherFunction = myObj.whatsThis;  
someOtherObject.anotherFunction();    // → someOtherObject. Called from the execution context of someOtherObject  
  
// Create a global variable whose value is a reference to the whatsThis method of object myObj  
var aGlobalVariable = myObj.whatsThis;  
aGlobalVariable();                    // → window. Called from the global execution context  
  
// And just to prove that nothing has changed in the original object...  
myObj.whatsThis();                    // → myObj. Called from the execution context of myObj
```

Lexical Scope: Deliberate Alteration of the Meaning of this

“Delegation” is a technique by which you can implement dynamic inheritance. This means you can call a method in some object and, for the duration of that method’s execution, you can deliberately alter the value of **this**.

The methods `Function.apply()` and `Function.call()` are used here and both take as their first parameter, the object to which **this** should refer.

```
var myObj = {  
  whatsThis : function() {  
    return this.a + this.b;  
  }  
}  
  
// Create some other objects that will be used to define the meaning of 'this'  
var obj1 = { a:1, b:2 };  
var obj2 = { a:3, b:4 };
```

Lexical Scope: Deliberate Alteration of the Meaning of this

“Delegation” is a technique by which you can implement dynamic inheritance. This means you can call a method in some object and, for the duration of that method’s execution, you can deliberately alter the value of **this**.

The methods `Function.apply()` and `Function.call()` are used here and both take as their first parameter, the object to which **this** should refer.

```
var myObj = {  
  whatsThis : function() {  
    return this.a + this.b;  
  }  
}  
  
// Create some other objects that will be used to define the meaning of 'this'  
var obj1 = { a:1, b:2 };  
var obj2 = { a:3, b:4 };  
  
// Invoke method whatsThis using different objects for the value of 'this'  
myObj.whatsThis.apply(obj1);    // → 3 because obj1 provides the value for 'this'  
myObj.whatsThis.call(obj2);     // → 7 because obj2 provides the value for 'this'
```


Lexical Scope: Deliberate Alteration of the Meaning of this

“Delegation” is a technique by which you can implement dynamic inheritance. This means you can call a method in some object and, for the duration of that method’s execution, you can deliberately alter the value of **this**.

The methods `Function.apply()` and `Function.call()` are used here and both take as their first parameter, the object to which **this** should refer.

```
var myObj = {  
  whatsThis : function() {  
    return this.a + this.b;  
  }  
}  
  
// Create some other objects that will be used to define the meaning of 'this'  
var obj1 = { a:1, b:2 };  
var obj2 = { a:3, b:4 };  
  
// Invoke method whatsThis using different objects for the value of 'this'  
myObj.whatsThis.apply(obj1);    // → 3 because obj1 provides the value for 'this'  
myObj.whatsThis.call(obj2);    // → 7 because obj2 provides the value for 'this'
```

Within each execution context of method `whatsThis()`, the value of **this** is temporarily reassigned to the object passed as the first parameter to either `apply()` or `call()`.

Lexical Scope: Variables Declared Inside Functions 1/2

When one function (known as the “outer” function) has another function (known as the “inner” function) declared within it, the scope of the outer function is **always** accessible to the inner function.

This is an example of ‘scope chaining’ and it applies even if the “outer” function is the Global Context.

```
var person = function() {  
  var firstName = "Harry";  
  var lastName  = "Hawk";  
}
```

Global Context: window

document

parent

top

localStorage

sessionStorage

location

history

etc...

Lexical Scope: Variables Declared Inside Functions 1/2

When one function (known as the “outer” function) has another function (known as the “inner” function) declared within it, the scope of the outer function is **always** accessible to the inner function. This is an example of ‘scope chaining’ and it applies even if the “outer” function is the Global Context.

```
var person = function() {  
  var firstName = "Harry";  
  var lastName  = "Hawk";  
}
```

All variables and properties of the Global Context are visible to the coding running inside function person.

Global Context: window

Function Context: person()

firstName lastName

document

parent

top

localStorage

sessionStorage

location

history

etc...

Lexical Scope: Variables Declared Inside Functions 2/2

The scopes of two sibling functions are not visible to each other, neither can code running in the scope of an outer function gain direct access to the variables within the scope of its inner functions.

```
var person = function() {  
  var firstName = "Harry";  
  var lastName = "Hawk";  
}
```

```
var vehicle = function() {  
  var make = "Honda";  
  var model = "Civic";  
}
```

Global Context: window

Function Context: person()

firstName

lastName

Function Context: vehicle()

make

model

document

parent

top

localStorage

sessionStorage

location

history

etc...



Declarations

Use of the **return** statement in functions

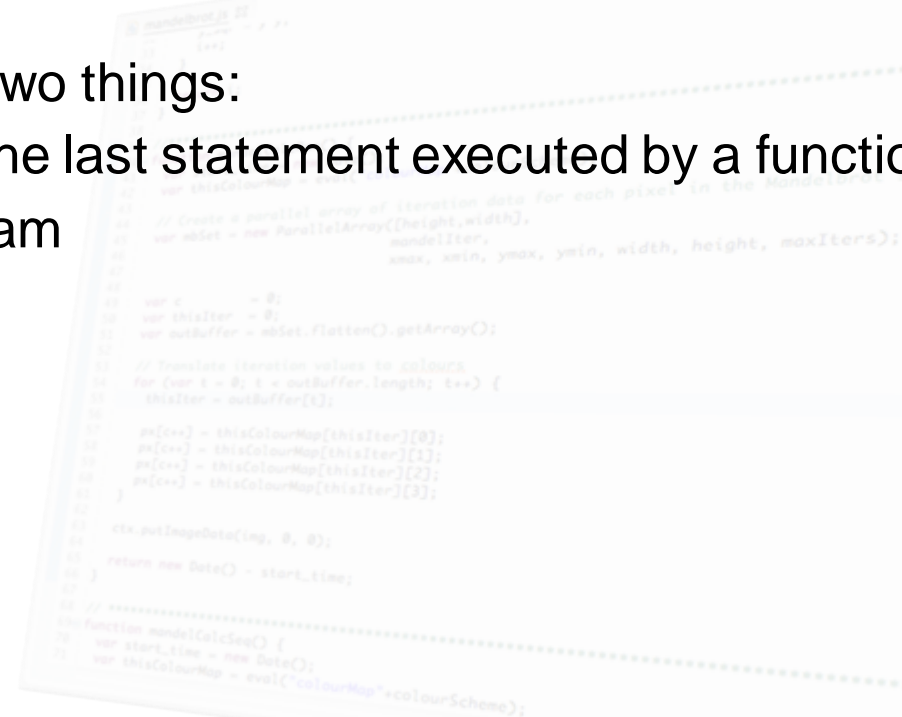
Using the **return** Statement

Generally speaking, a function is called because you require some value to be calculated that will then take the place of the function call in your sequence of instructions. (This is known as “referential transparency”, but more about that later).

JavaScript uses the keyword **return** to define two things:

- An exit point. If present, **return** is **always** the last statement executed by a function.
- The value to be returned to the calling program

```
function doSomething(a,b) {  
  if (typeof a === 'number' &&  
      typeof b === 'number') {  
    return a * b;  
  }  
  else {  
    return a + b;  
  }  
}
```



```
function doSomething(a,b) {  
  if (typeof a === 'number' &&  
      typeof b === 'number') {  
    return a * b;  
  }  
  else {  
    return a + b;  
  }  
}
```

Using the **return** Statement

Generally speaking, a function is called because you require some value to be calculated that will then take the place of the function call in your sequence of instructions. (This is known as “referential transparency”, but more about that later).

JavaScript uses the keyword **return** to define two things:

- An exit point. If present, **return** is **always** the last statement executed by a function.
- The value to be returned to the calling program

```
function doSomething(a,b) {  
  if (typeof a === 'number' &&  
      typeof b === 'number') {  
    return a * b;  
  }  
  else {  
    return a + b;  
  }  
}
```

Whilst it is perfectly possible for a function to use multiple return statements, this should be avoided as a matter of good coding practice.

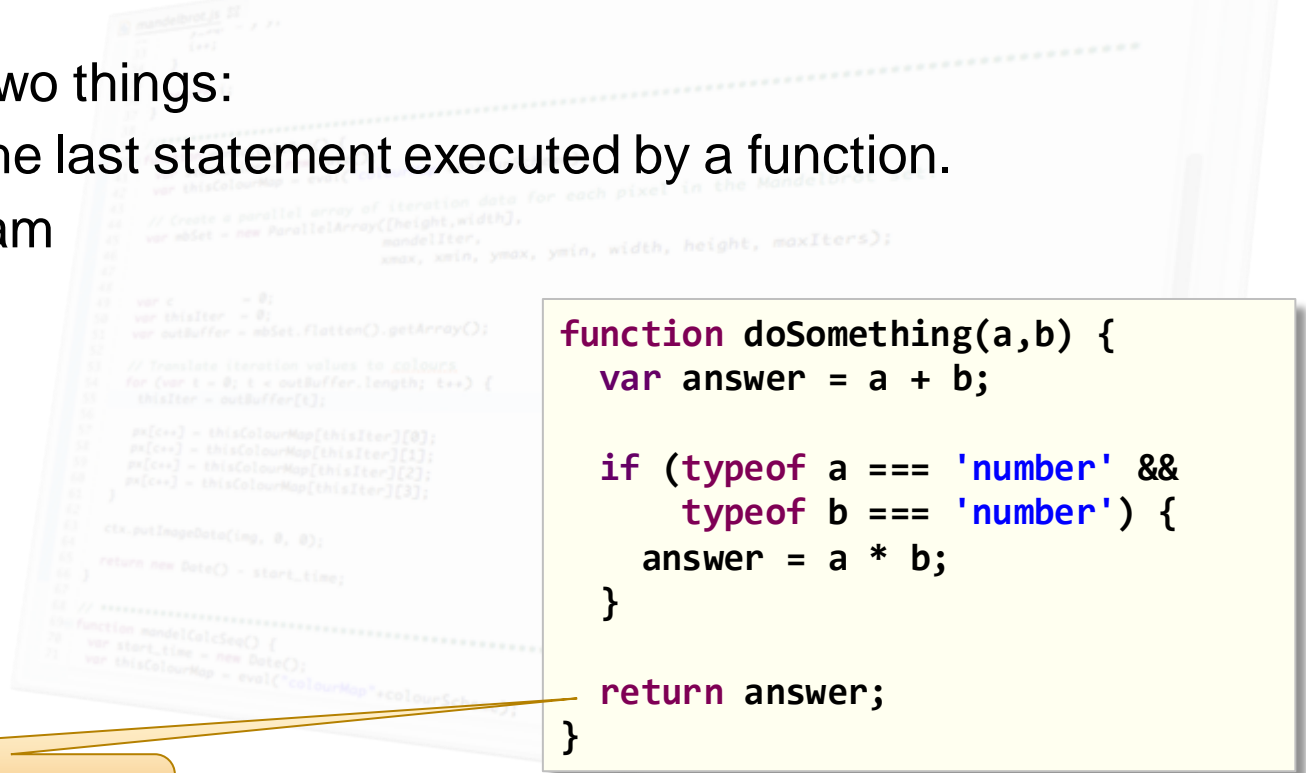
In terms of unit testing and quality assurance of code, a function should only ever have **one** exit point.

Using the **return** Statement

Generally speaking, a function is called because you require some value to be calculated that will then take the place of the function call in your sequence of instructions. (This is known as “referential transparency”, but more about that later).

JavaScript uses the keyword **return** to define two things:

- An exit point. If present, **return** is **always** the last statement executed by a function.
- The value to be returned to the calling program



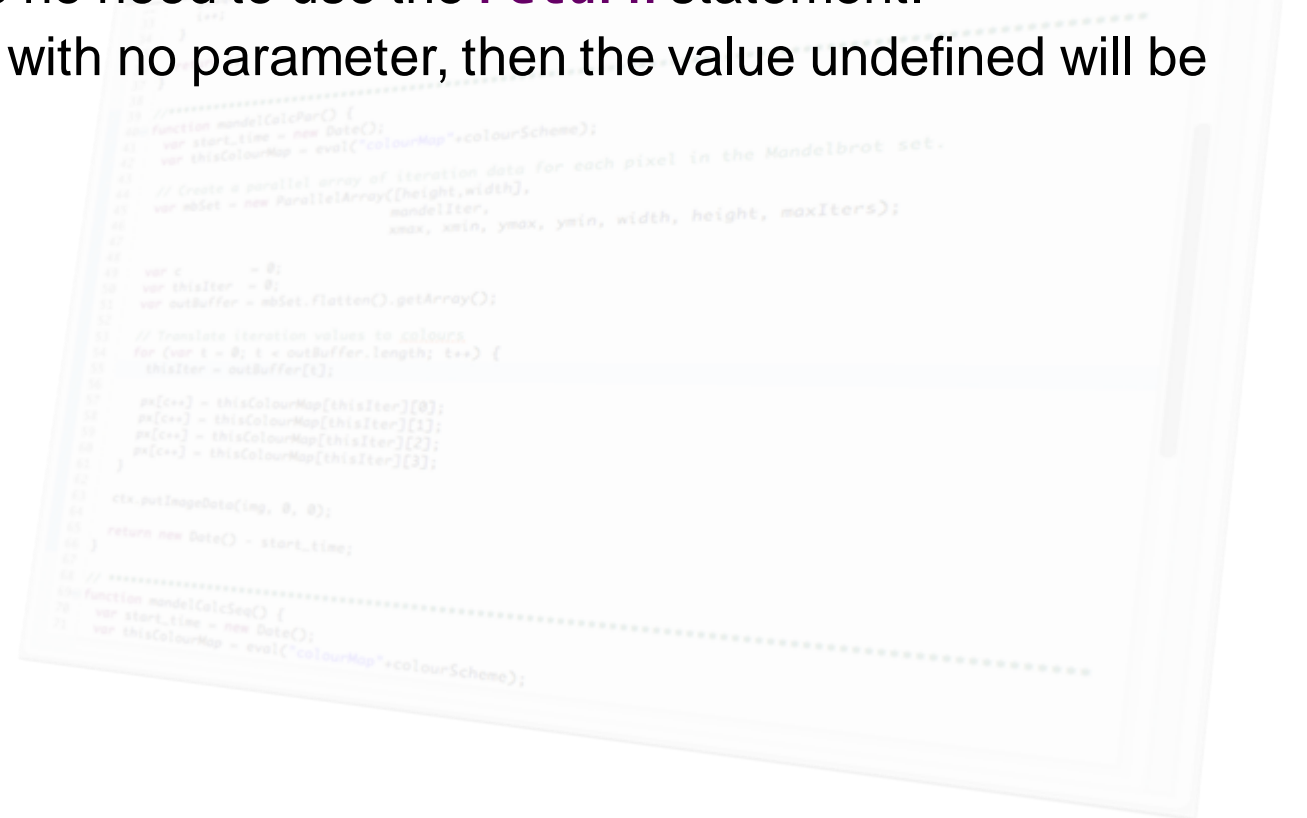
```
function doSomething(a,b) {  
    var answer = a + b;  
  
    if (typeof a === 'number' &&  
        typeof b === 'number') {  
        answer = a * b;  
    }  
  
    return answer;  
}
```

This coding structure is preferable because the function now has one and only one exit point.

Functions where **return** isn't needed

Some functions perform tasks that specifically rely on side-effect behaviour. Therefore, the desired outcome of calling the function is the behaviour of the side-effect, rather than some explicit value returned from the function. In this case, there is no need to use the **return** statement.

If the **return** keyword is either missing or used with no parameter, then the value undefined will be returned to the calling program.



Functions where **return** isn't needed

Some functions perform tasks that specifically rely on side-effect behaviour. Therefore, the desired outcome of calling the function is the behaviour of the side-effect, rather than some explicit value returned from the function. In this case, there is no need to use the **return** statement.

If the **return** keyword is either missing or used with no parameter, then the value undefined will be returned to the calling program.

```
function callLater(fn, args, ctx) {  
  setTimeout(function() { fn.apply(ctx,args); }, 2000);  
}
```

In this case, the `callLater()` function is being called because it has desirable side-effects – it will call the specified function after a delay of 2 seconds.

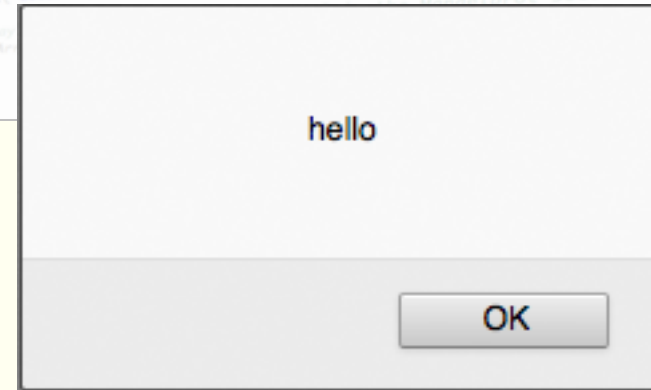
Functions where **return** isn't needed

Some functions perform tasks that specifically rely on side-effect behaviour. Therefore, the desired outcome of calling the function is the behaviour of the side-effect, rather than some explicit value returned from the function. In this case, there is no need to use the **return** statement.

If the **return** keyword is either missing or used with no parameter, then the value undefined will be returned to the calling program.

```
function callLater(fn, args, ctx) {  
    setTimeout(function() { fn.apply(ctx, args); }, 2000);  
}
```

```
var value = callLater(alert, ['hello']);
```



(After a 2 second delay)

In this case, the `callLater()` function is being called because it has desirable side-effects – it will call the specified function after a delay of 2 seconds.

Functions where **return** isn't needed

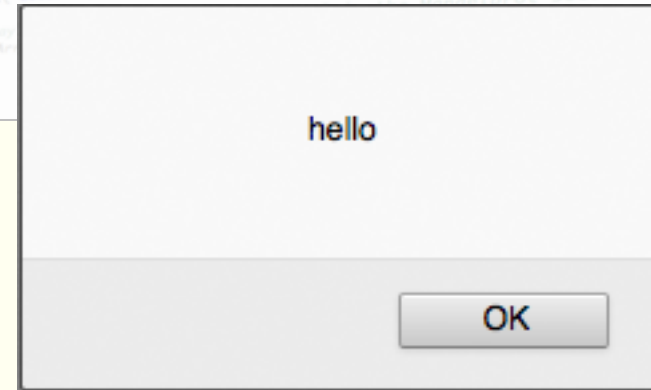
Some functions perform tasks that specifically rely on side-effect behaviour. Therefore, the desired outcome of calling the function is the behaviour of the side-effect, rather than some explicit value returned from the function. In this case, there is no need to use the **return** statement.

If the **return** keyword is either missing or used with no parameter, then the value **undefined** will be returned to the calling program.

```
function callLater(fn, args, ctx) {  
    setTimeout(function() { fn.apply(ctx, args); }, 2000);  
}
```

```
var value = callLater(alert, ['hello']);
```

```
value; // → undefined
```



(After a 2 second delay)

In this case, the `callLater()` function is being called because it has desirable side-effects – it will call the specified function after a delay of 2 seconds.

In this particular case, the return value of function `callLater()` is not important.

Return Values from Functions: Returning **this** 1/2

It is also possible that a function returns a reference to its own execution scope.

For instance, if you are referencing an HTML button in a browser window, then although the following code is perfectly correct, it is somewhat inconvenient.

```
var myButton = jQuery('#myButton');  
  
myButton.text('Click here');  
myButton.css('color', '#F80');  
myButton.bind('click', function() { alert('Hi there!'); });
```

Return Values from Functions: Returning **this** 2/2

Frameworks such as jQuery and SAPUI5 are setup so that functions handling DOM elements return a reference to their own execution scope.

In other words, these functions return a reference to the DOM element currently being manipulated, identified by the variable **this**.

Returning **this** allows you to chain method calls together in a technique know either as “cascading” or “method chaining”.

```
jQuery('#myButton')  
  .text('Click here')  
  .css('color', '#F80')  
  .bind('click', function() { alert('Hi there!'); });
```




Declarations

Scope Chaining and Variable Lifespan

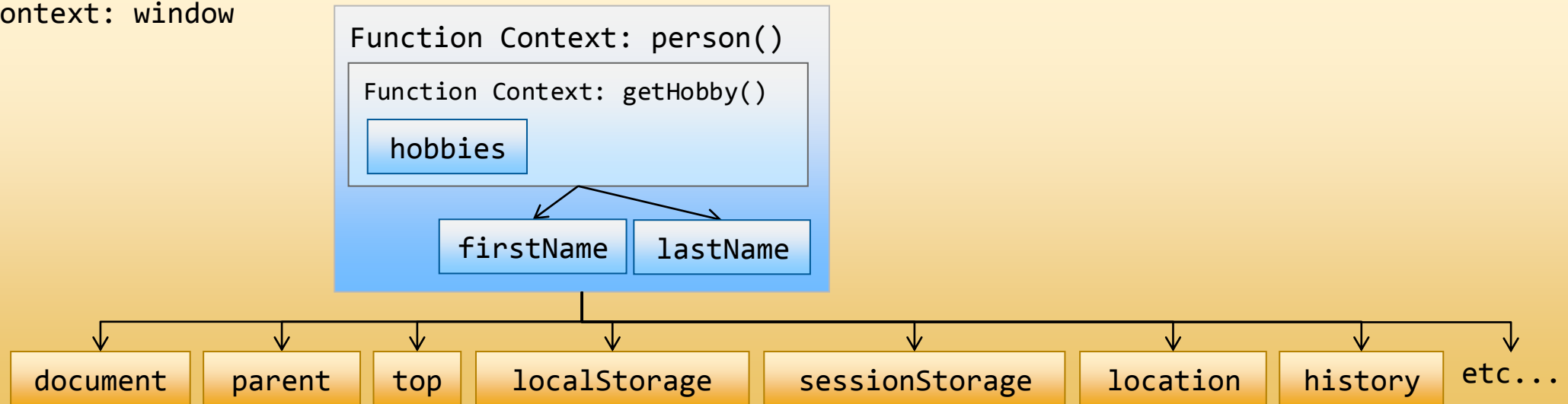
Scope Chaining 1/2

Within the person object, we have now created an inner function called `getHobby()`.

Through scope chaining, the coding in `getHobby()` can access the variables `firstName` and `lastName`, but the coding in `person()` cannot directly access the `hobbies` array.

```
var person = function() {  
  var firstName = "Harry";  
  var lastName = "Hawk";  
  
  var getHobby = function(n) {  
    var hobbies = ["swimming", "cycling"];  
    return firstName + " likes " + hobbies[n];  
  }  
}
```

Global Context: window



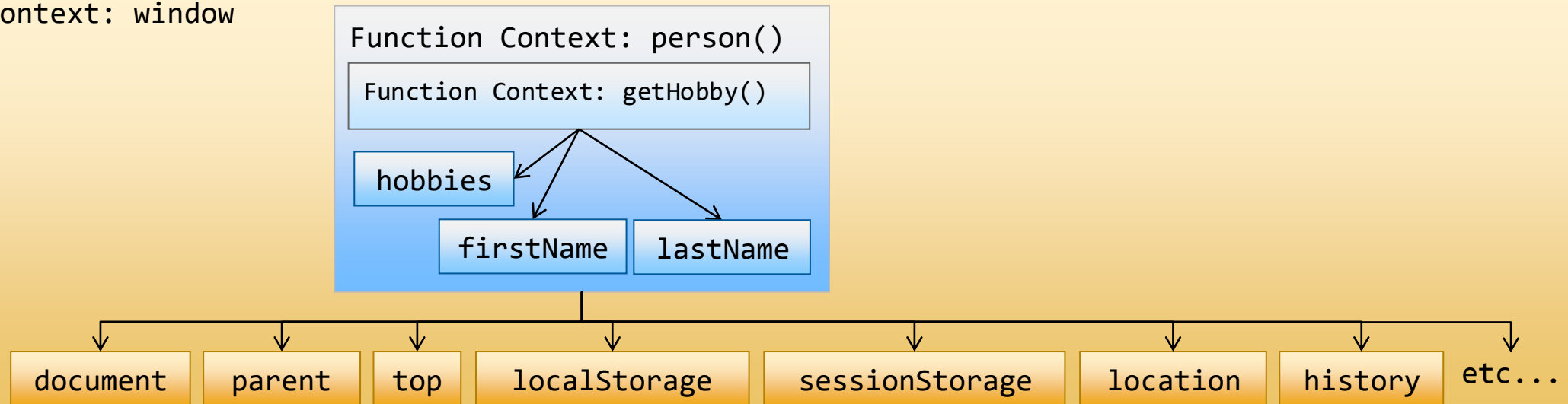
Scope Chaining 2/2

From a software design perspective, this is not a good situation since the coding in function `person()` will probably need access to the `hobbies` array.

However, scope chaining allows us to move the declaration of the `hobbies` array out of the `getHobby()` function and into the `person()` function without any problem.

```
var person = function() {  
  var firstName = "Harry";  
  var lastName = "Hawk";  
  var hobbies = ["swimming", "cycling"];  
  
  var getHobby = function(n) {  
    return firstName + " likes " + hobbies[n];  
  }  
}
```

Global Context: window



Scope Chaining and Variable Lifespan 1/3

All variables declared inside a function are private to that function's scope, making them invisible to the code that called the function.

```
var someGuy = (function () {  
  var firstName = "Harry";  
  var lastName  = "Hawk";  
  var hobbies   = [ "swimming", "cycling" ];  
})
```

The variables `firstName`, `lastName` and `hobbies` are visible only within the scope of this anonymous function.

This anonymous function is invoked automatically.

}) () ;

Scope Chaining and Variable Lifespan 1/3

All variables declared inside a function are private to that function's scope, making them invisible to the code that called the function.

However, these private variables can be exposed by returning an object that references them.

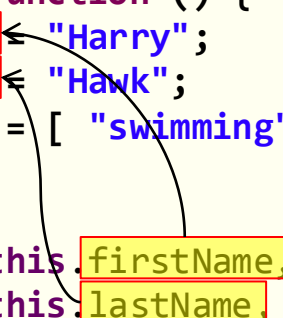
```
var someGuy = (function () {  
    var firstName = "Harry";  
    var lastName  = "Hawk";  
    var hobbies   = [ "swimming", "cycling" ];  
  
    return {  
        firstName: this.firstName,  
        lastName  : this.lastName,  
        getHobby  : function(n) { return hobbies[n]; },  
        setHobby  : function(h) { hobbies.push(h); }  
    }  
})();
```

Scope Chaining and Variable Lifespan 1/3

All variables declared inside a function are private to that function's scope, making them invisible to the code that called the function.

However, these private variables can be exposed by returning an object that references them.

```
var someGuy = (function () {  
  var firstName ← "Harry";  
  var lastName ← "Hawk";  
  var hobbies = [ "swimming", "cycling" ];  
  
  return {  
    firstName: this.firstName,  
    lastName : this.lastName,  
    getHobby : function(n) { return hobbies[n]; },  
    setHobby : function(h) { hobbies.push(h); }  
  }  
})();
```



Scope Chaining and Variable Lifespan 1/3

All variables declared inside a function are private to that function's scope, making them invisible to the code that called the function.

However, these private variables can be exposed by returning an object that references them.

```
var someGuy = (function () {  
    var firstName = "Harry";  
    var lastName  = "Hawk";  
    var hobbies   = [ "swimming", "cycling" ];  
  
    return {  
        firstName: this.firstName,  
        lastName  : this.lastName,  
        getHobby  : function(n) { return hobbies[n]; },  
        setHobby  : function(h) { hobbies.push(h); }  
    }  
})();  
  
someGuy.firstName;    // → Harry  
someGuy.lastName;     // → Hawk
```


Scope Chaining and Variable Lifespan 2/3

In addition to returning a simple reference, scope chaining allows you to return an inner function that references a variable in the scope of the outer function.

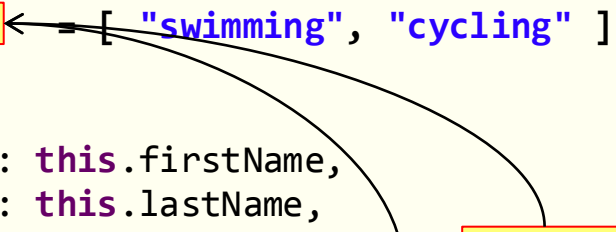
```
var someGuy = (function () {  
    var firstName = "Harry";  
    var lastName  = "Hawk";  
    var hobbies   = [ "swimming", "cycling" ];  
  
    return {  
        firstName: this.firstName,  
        lastName  : this.lastName,  
        getHobby  : function(n) { return hobbies[n]; },  
        setHobby  : function(h) { hobbies.push(h); }  
    }  
})();
```

Scope Chaining and Variable Lifespan 2/3

In addition to returning a simple reference, scope chaining allows you to return an inner function that references a variable in the scope of the outer function.

In this example, methods `getHobby` and `setHobby` of the returned object are inner functions that reference the `hobbies` variable in the scope of the outer anonymous function.

```
var someGuy = (function () {  
  var firstName = "Harry";  
  var lastName = "Hawk";  
  var hobbies = [ "swimming", "cycling" ];  
  
  return {  
    firstName: this.firstName,  
    lastName : this.lastName,  
    getHobby : function(n) { return hobbies[n]; },  
    setHobby : function(h) { hobbies.push(h); }  
  }  
})();
```

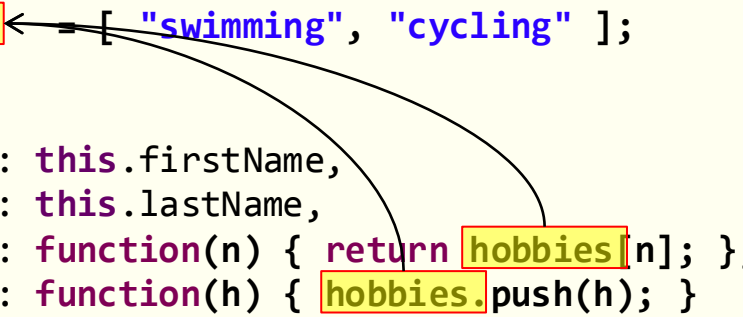


Scope Chaining and Variable Lifespan 2/3

In addition to returning a simple reference, scope chaining allows you to return an inner function that references a variable in the scope of the outer function.

In this example, methods `getHobby` and `setHobby` of the returned object are inner functions that reference the `hobbies` variable in the scope of the outer anonymous function.

```
var someGuy = (function () {  
  var firstName = "Harry";  
  var lastName = "Hawk";  
  var hobbies = [ "swimming", "cycling" ];  
  
  return {  
    firstName: this.firstName,  
    lastName : this.lastName,  
    getHobby : function(n) { return hobbies[n]; },  
    setHobby : function(h) { hobbies.push(h); }  
  }  
})();  
  
someGuy.getHobby(1);      // → "cycling"  
someGuy.setHobby("hiking"); // → [ "swimming", "cycling", "hiking"]
```



Scope Chaining and Variable Lifespan 3/3

You might look at this coding and wonder why it works; after all, when a function terminates, don't all the variables within that function's scope cease to exist?

Here we're using an anonymous function to generate a person object. The returned object is stored in a variable called `someGuy`, then the anonymous function terminates and its scope disappears...

```
var someGuy = (function () {
  var firstName = "Harry";
  var lastName  = "Hawk";
  var hobbies   = [ "swimming", "cycling" ];

  return {
    firstName: this.firstName,
    lastName : this.lastName,
    getHobby : function(n) { return hobbies[n]; },
    setHobby : function(h) { hobbies.push(h); }
  }
})();

someGuy.getHobby(1);    // → "cycling"
someGuy.setHobby("hiking"); // → [ "swimming", "cycling", "hiking"]
```

Scope Chaining and Variable Lifespan 3/3

You might look at this coding and wonder why it works; after all, when a function terminates, don't all the variables within that function's scope cease to exist?

Here we're using an anonymous function to generate a person object. The returned object is stored in a variable called `someGuy`, then the anonymous function terminates and its scope disappears...

```
var someGuy = (function () {  
    var firstName = "Harry";  
    var lastName  = "Hawk";  
    var hobbies   = [ "swimming", "cycling" ];  
  
    return {  
        firstName: this.firstName,  
        lastName  : this.lastName,  
        getHobby  : function(n) { return hobbies[n]; },  
        setHobby  : function(h) { hobbies.push(h); }  
    }  
})();
```

```
someGuy.getHobby(1);    // → "cycling"  
someGuy.setHobby("hiking"); // → [ "swimming", "cycling", "hiking"]
```

Yet, in spite of this apparent contradiction, the values of the supposedly terminated function are still alive and well...

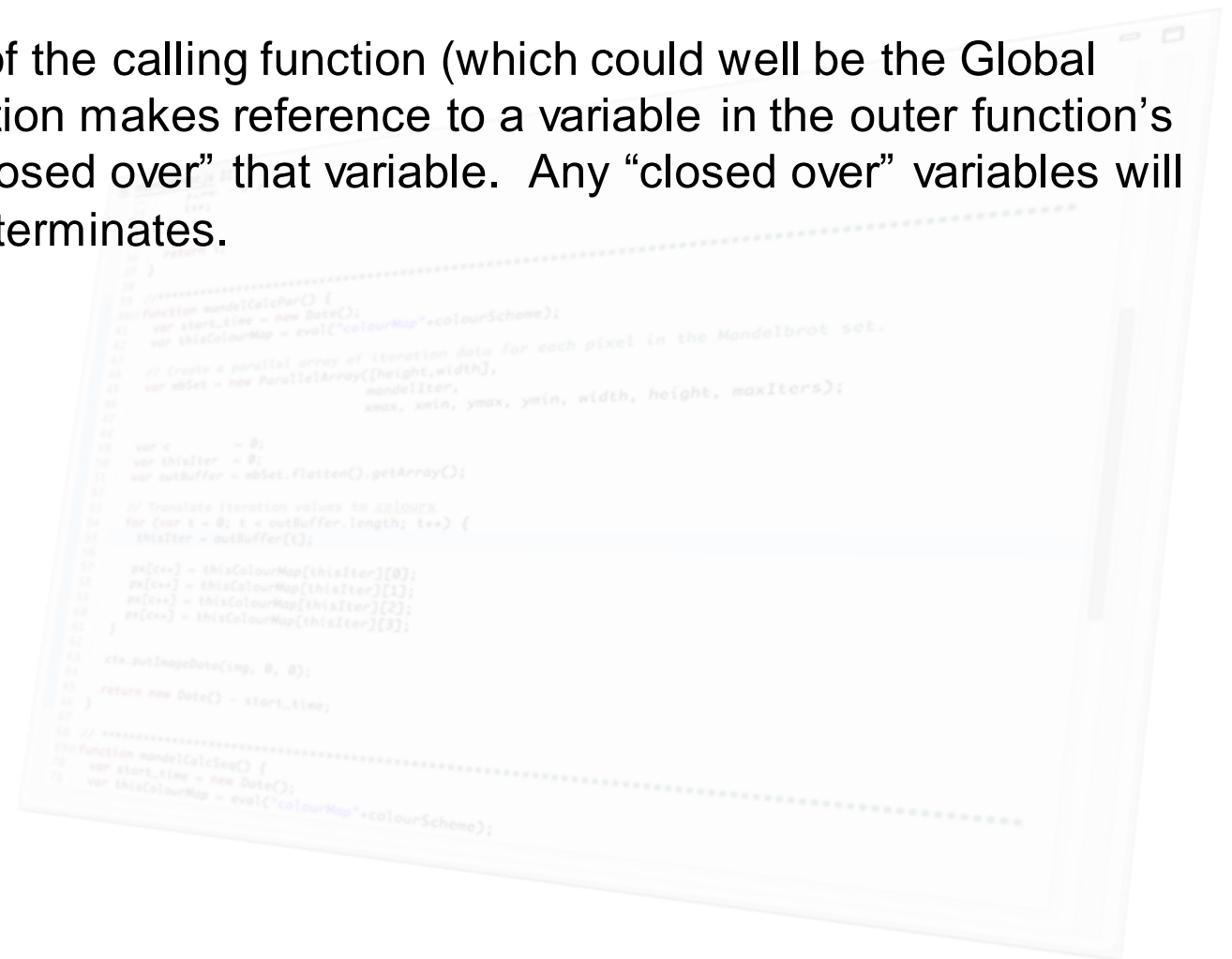
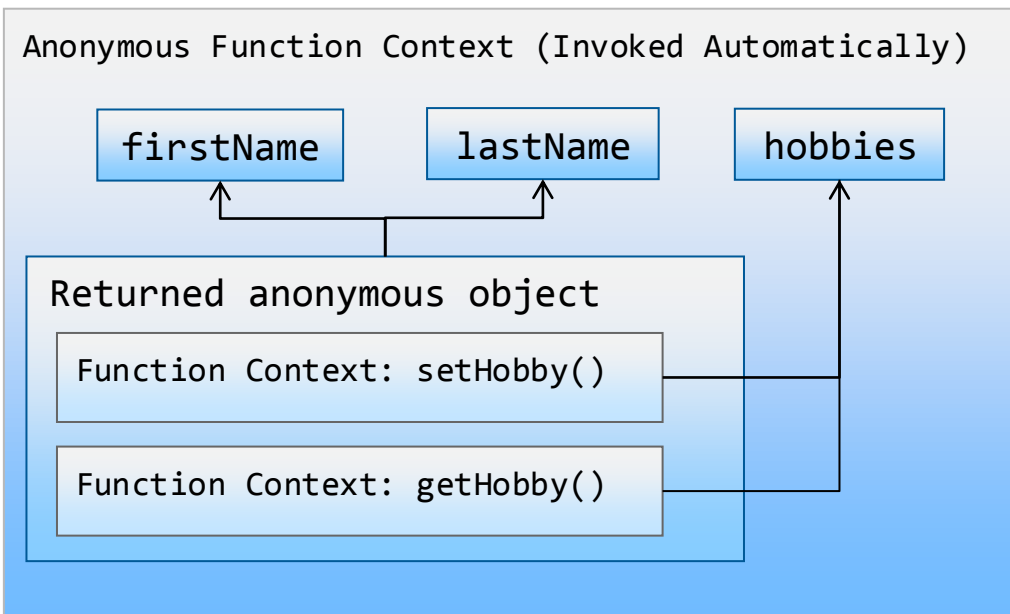


Declarations

Creating Public and Private Object Properties

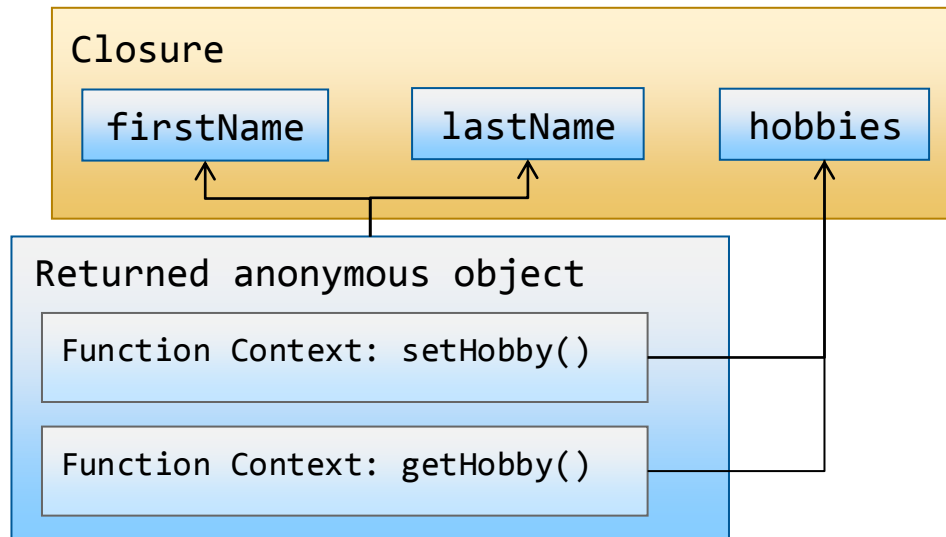
Creating Public and Private Object Properties: Closure

As soon as a function is called, the lexical scope of the calling function (which could well be the Global Context) must be preserved. When an inner function makes reference to a variable in the outer function's lexical scope, the inner function is said to have “closed over” that variable. Any “closed over” variables will not be garbage collected when the outer function terminates.

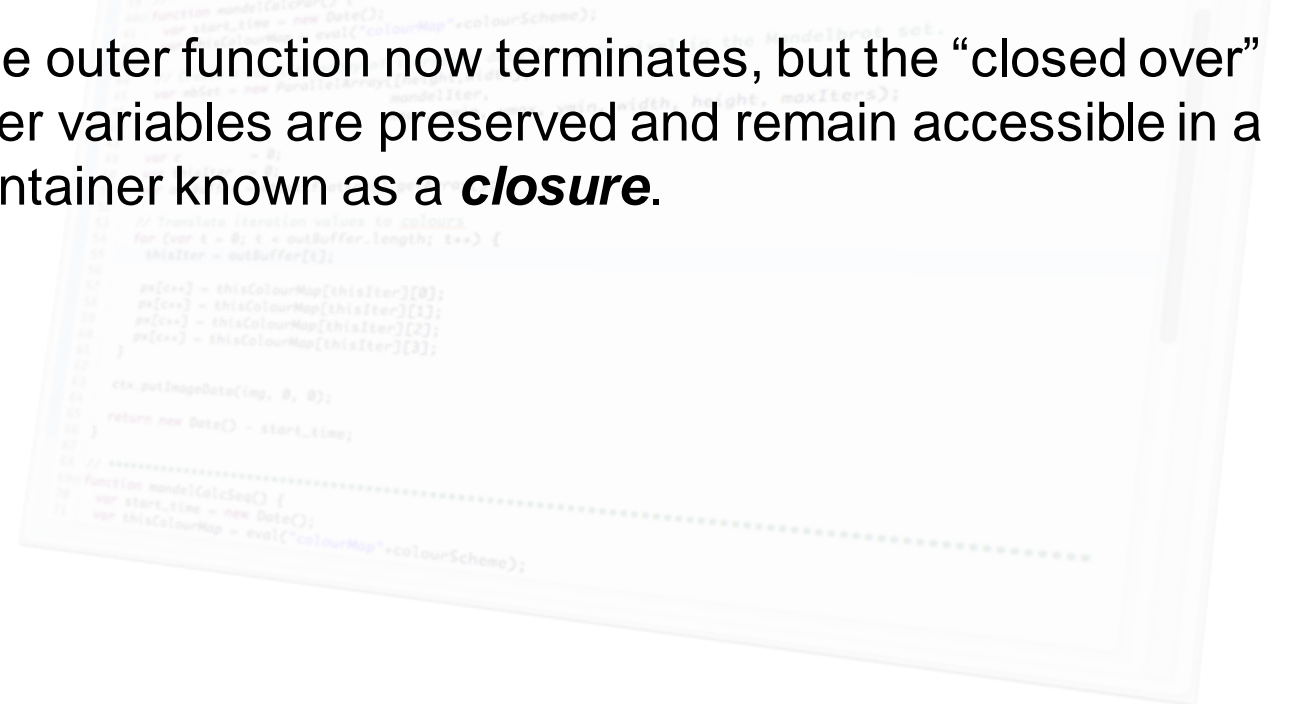


Creating Public and Private Object Properties: Closure

As soon as a function is called, the lexical scope of the calling function (which could well be the Global Context) must be preserved. When an inner function makes reference to a variable in the outer function's lexical scope, the inner function is said to have “closed over” that variable. Any “closed over” variables will not be garbage collected when the outer function terminates.

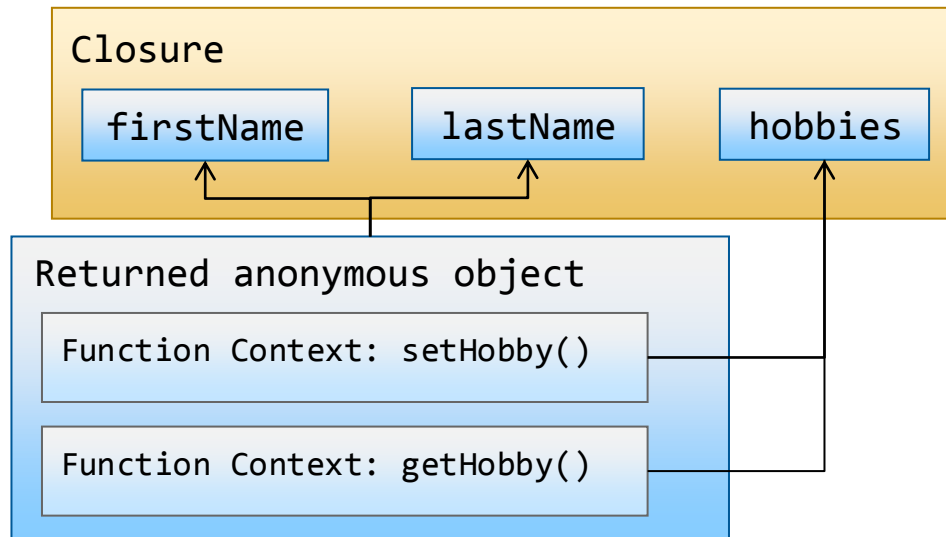


The outer function now terminates, but the “closed over” over variables are preserved and remain accessible in a container known as a ***closure***.



Creating Public and Private Object Properties: Closure

As soon as a function is called, the lexical scope of the calling function (which could well be the Global Context) must be preserved. When an inner function makes reference to a variable in the outer function's lexical scope, the inner function is said to have “closed over” that variable. Any “closed over” variables will not be garbage collected when the outer function terminates.



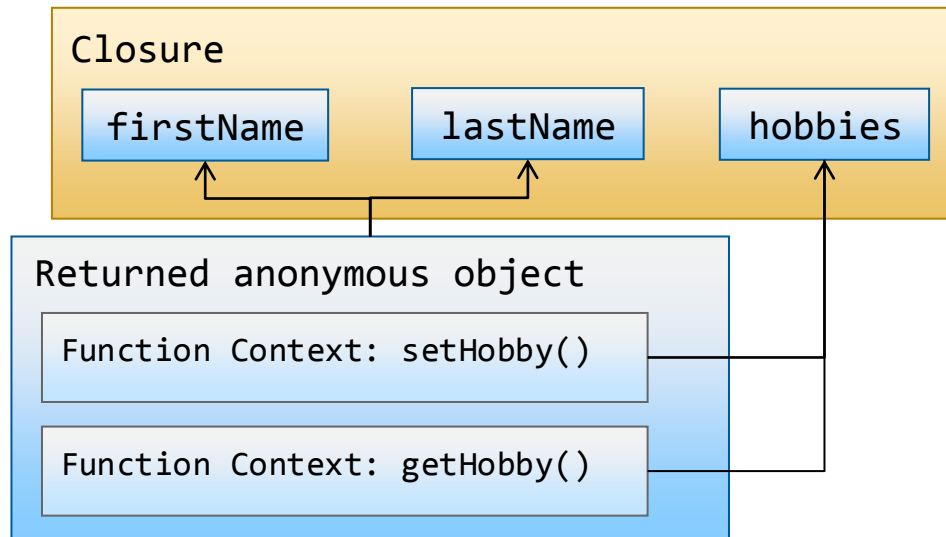
The outer function now terminates, but the “closed over” over variables are preserved and remain accessible in a container known as a ***closure***.

The use of closures is considered a fundamental design technique in many scripting languages such as JavaScript, Python, Ruby and SmallTalk, or functional programming languages such as Haskell and OCaml.

Closures are also possible in Java by means of anonymous inner functions.

Creating Public and Private Object Properties: Closure

As soon as a function is called, the lexical scope of the calling function (which could well be the Global Context) must be preserved. When an inner function makes reference to a variable in the outer function's lexical scope, the inner function is said to have “closed over” that variable. Any “closed over” variables will not be garbage collected when the outer function terminates.



The outer function now terminates, but the “closed over” over variables are preserved and remain accessible in a container known as a ***closure***.

The use of closures is considered a fundamental design technique in many scripting languages such as JavaScript, Python, Ruby and SmallTalk, or functional programming languages such as Haskell and OCaml.

Closures are also possible in Java by means of anonymous inner functions.

ABAP has no concept of closures...

Closure and The Module Pattern 1/3

Remember that in JavaScript, there is no explicit mechanism to define the visibility of an object property. However, we often find ourselves needing to create an object in which some properties are public and some are private (accessible only through getter and setter methods).

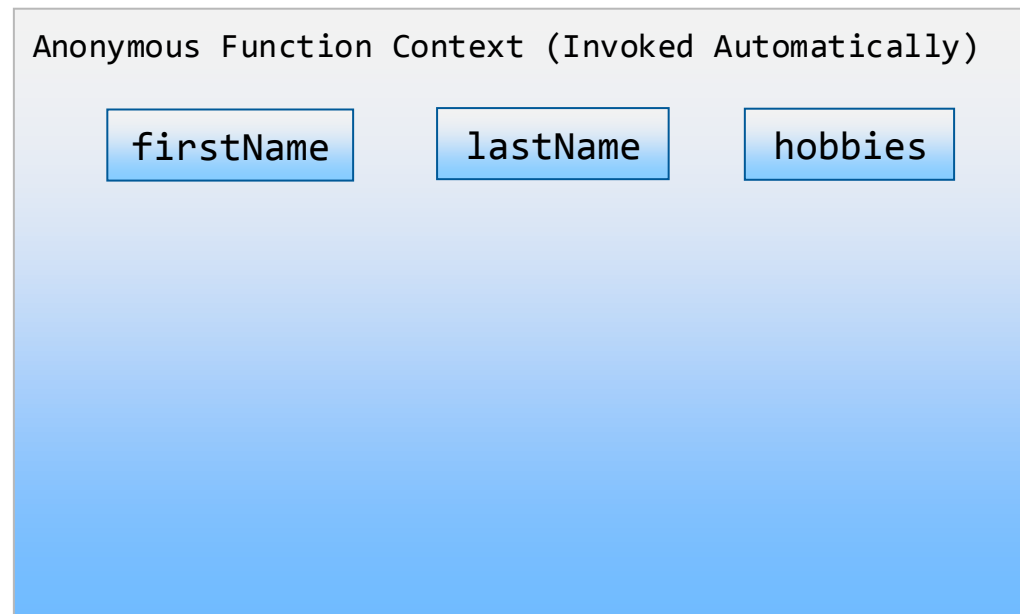
The "Module Pattern" is where a closure is used to act as a container for these "private" properties.

```
var someGuy = (function () {  
    var firstName = "Harry";  
    var lastName  = "Hawk";  
    var hobbies   = [ "swimming", "cycling" ];  
  
    })();
```

Closure and The Module Pattern 1/3

Remember that in JavaScript, there is no explicit mechanism to define the visibility of an object property. However, we often find ourselves needing to create an object in which some properties are public and some are private (accessible only through getter and setter methods).

The "Module Pattern" is where a closure is used to act as a container for these "private" properties.

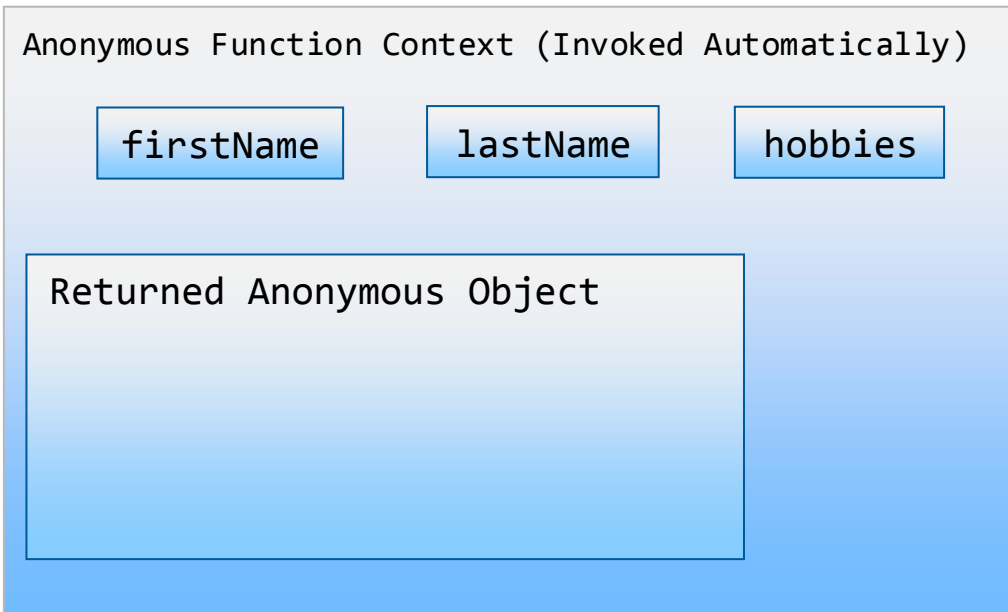


```
var someGuy = (function () {  
    var firstName = "Harry";  
    var lastName = "Hawk";  
    var hobbies = [ "swimming", "cycling" ];  
  
})();
```

The execution scope of the outer function will become the closure that contains those variables we wish to keep private.

Closure and The Module Pattern 2/3

The outer function then returns an anonymous object.

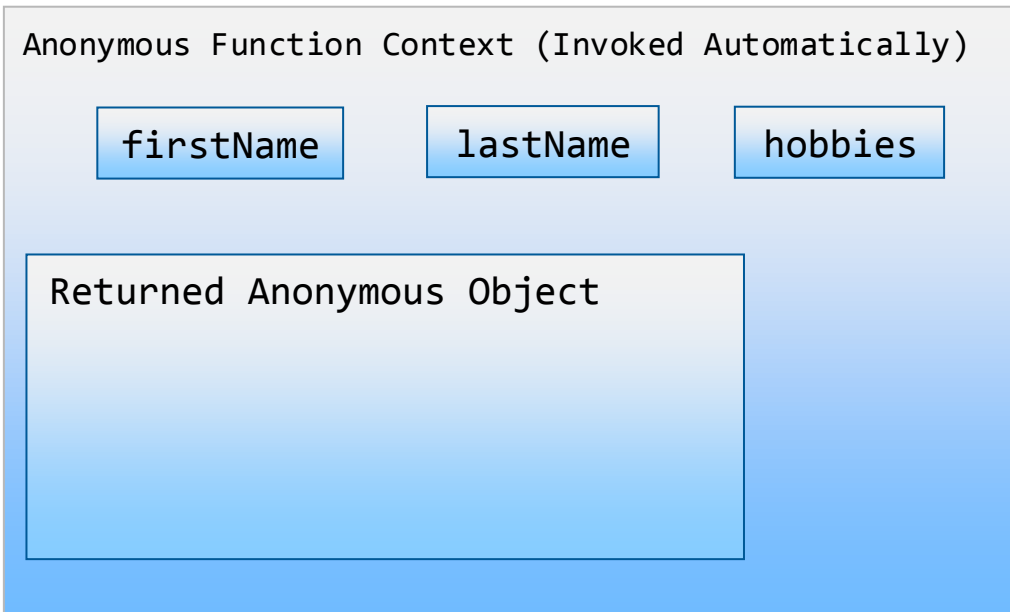


```
var person = (function () {  
    var firstName = "Harry";  
    var lastName  = "Hawk";  
    var hobbies   = [ "swimming", "cycling" ];  
  
    return {  
  
    }  
})();
```

Closure and The Module Pattern 2/3

The outer function then returns an anonymous object.

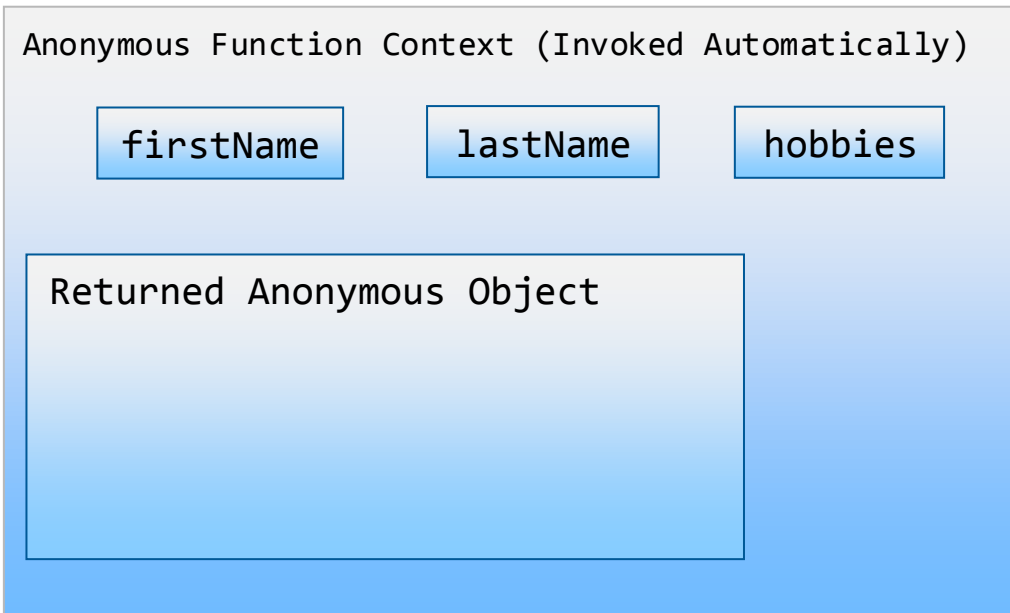
The properties of this anonymous object “close over” the variables in the outer function’s scope.



```
var person = (function () {  
  var firstName ← "Harry";  
  var lastName ← "Hawk";  
  var hobbies   = [ "swimming", "cycling" ];  
  
  return {  
    firstName: this.firstName,  
    lastName : this.lastName,  
  
  }  
})();
```


Closure and The Module Pattern 3/3

The anonymous object also returns two inner functions that both “close over” the same array in the outer function’s scope.



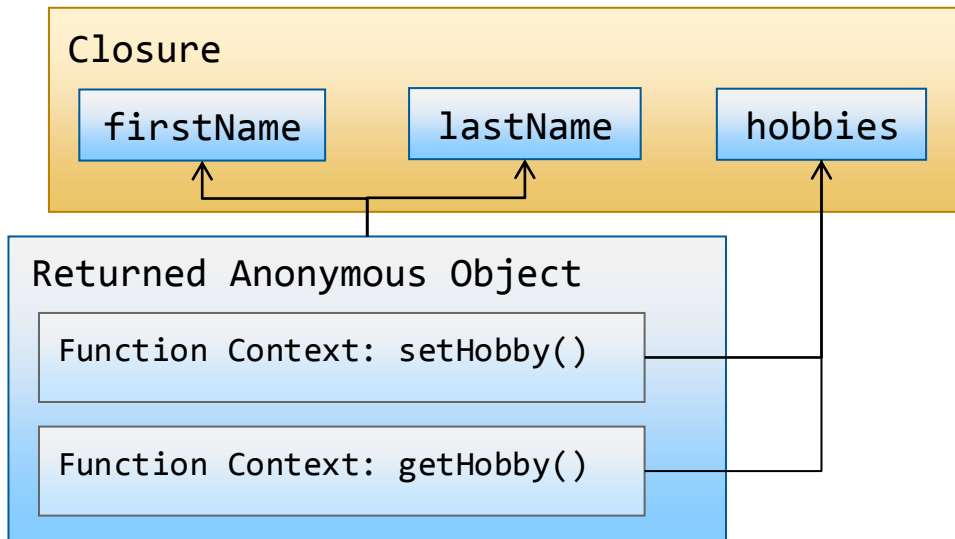
```
var person = (function () {  
  var firstName = "Harry";  
  var lastName = "Hawk";  
  var hobbies ← [ "swimming", "cycling" ];  
  
  return {  
    firstName: this.firstName,  
    lastName : this.lastName,  
    getHobby : function(n) { return hobbies[n]; },  
    setHobby : function(h) { hobbies.push(h); }  
  }  
})();
```

The diagram shows the JavaScript code for the Module Pattern. The variable `person` is assigned an anonymous function. Inside the function, `firstName` and `lastName` are declared, and `hobbies` is an array containing "swimming" and "cycling". The function returns an object with `firstName`, `lastName`, `getHobby`, and `setHobby` properties. Arrows indicate that the `getHobby` and `setHobby` functions close over the `hobbies` array.

Closure and The Module Pattern 3/3

The anonymous object also returns two inner functions that both “close over” the same array in the outer function’s scope.

When the outer function terminates, any “closed over” variables are not garbage collected. The closure acts as the container within which all our “private” variables live.



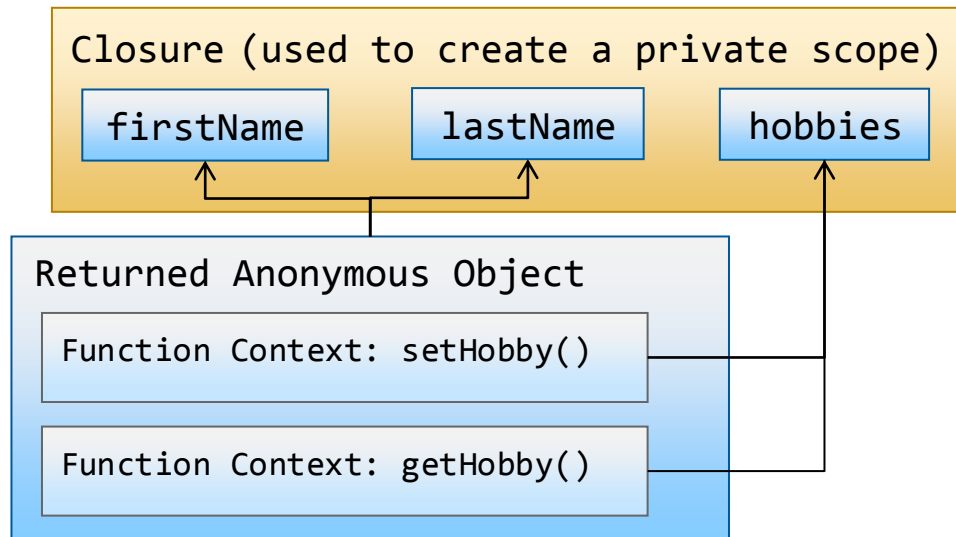
```
var person = (function () {
    var firstName = "Harry";
    var lastName = "Hawk";
    var hobbies = [ "swimming", "cycling" ];

    return {
        firstName: this.firstName,
        lastName : this.lastName,
        getHobby : function(n) { return hobbies[n]; },
        setHobby : function(h) { hobbies.push(h); }
    }
})();
```

Closure and The Module Pattern 3/3

The anonymous object also returns two inner functions that both “close over” the same array in the outer function’s scope.

When the outer function terminates, any “closed over” variables are not garbage collected. The closure acts as the container within which all our “private” variables live.



```
var person = (function () {  
    var firstName = "Harry";  
    var lastName = "Hawk";  
    var hobbies = [ "swimming", "cycling" ];  
  
    return {  
        firstName: this.firstName,  
        lastName : this.lastName,  
        getHobby : function(n) { return hobbies[n]; },  
        setHobby : function(h) { hobbies.push(h); }  
    }  
})();
```

So even though JavaScript has no explicit means of defining the visibility of object properties, we can use a closure as the container within which private properties are hidden.

Appendix: Busting Some Closure Myths

There are many misconceptions surrounding closures, and each item in the following list is wrong! Do not believe anyone who tries to convince you that:

- **Closures are only created by calling inner functions**

No. A closure is created as soon as a function is called. It does not matter whether an outer function calls an inner function, or a function is called from the global context: function calls are function calls and closures are created in both cases.

- **Closures are created only if an outer function returns an inner function**

No. A closure is created independently of whether the outer function use the **return** keyword or not

- **The value of "closed over" variables become fixed once a closure is created**

No. The scope of the outer function is preserved and any closed over variables are references to real objects, not references to static values

- **Closures can only be created by anonymous functions**

Hmmm, really? Go directly to jail. Do not pass Go. Do not collect £/\$/€ 200

- **Closures cause memory leaks**

Earlier versions of Internet Explorer were famous for their memory leaks and the use of closures usually got the blame. In reality, the usual culprit was a situation in which a function referenced a DOM element, and then an attribute of that same DOM element referenced a variable in the lexical scope of the function, creating a circular reference.

Its actually very difficult to create a circular references between a variable in a closure and the coding that references the closed over variables