

# JavaScript for ABAP Programmers

## Imperative vs. Functional Programming

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 1<sup>st</sup> class citizens

OO using referential inheritance

Imperative or Functional programming

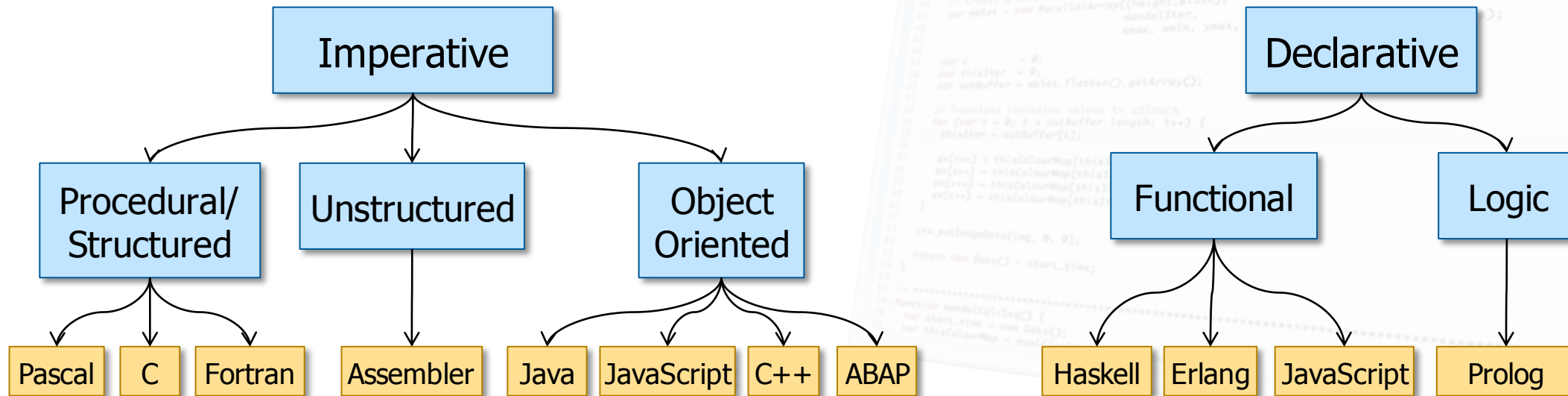


# Different Schools of Thought

# Programming in General 1/2

In broad terms, the world of computer programming is split into various, overlapping schools of thought. These schools of thought differ primarily on the approach used to design and build a computer program.

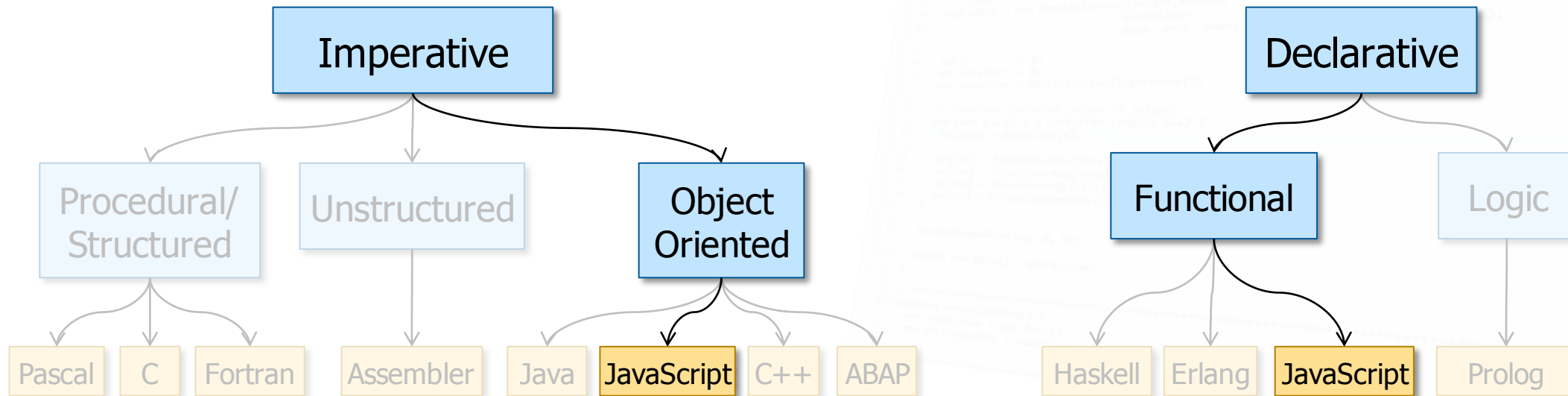
The (very incomplete) diagram below shows two of the main schools of thought in programming and some of their more widely known subdivisions.



# Programming in General 2/3

The divisions between these categories are not hard and fast, and some languages (like JavaScript) can belong to multiple categories.

Here we will be looking at using JavaScript in both the Imperative and Functional styles.



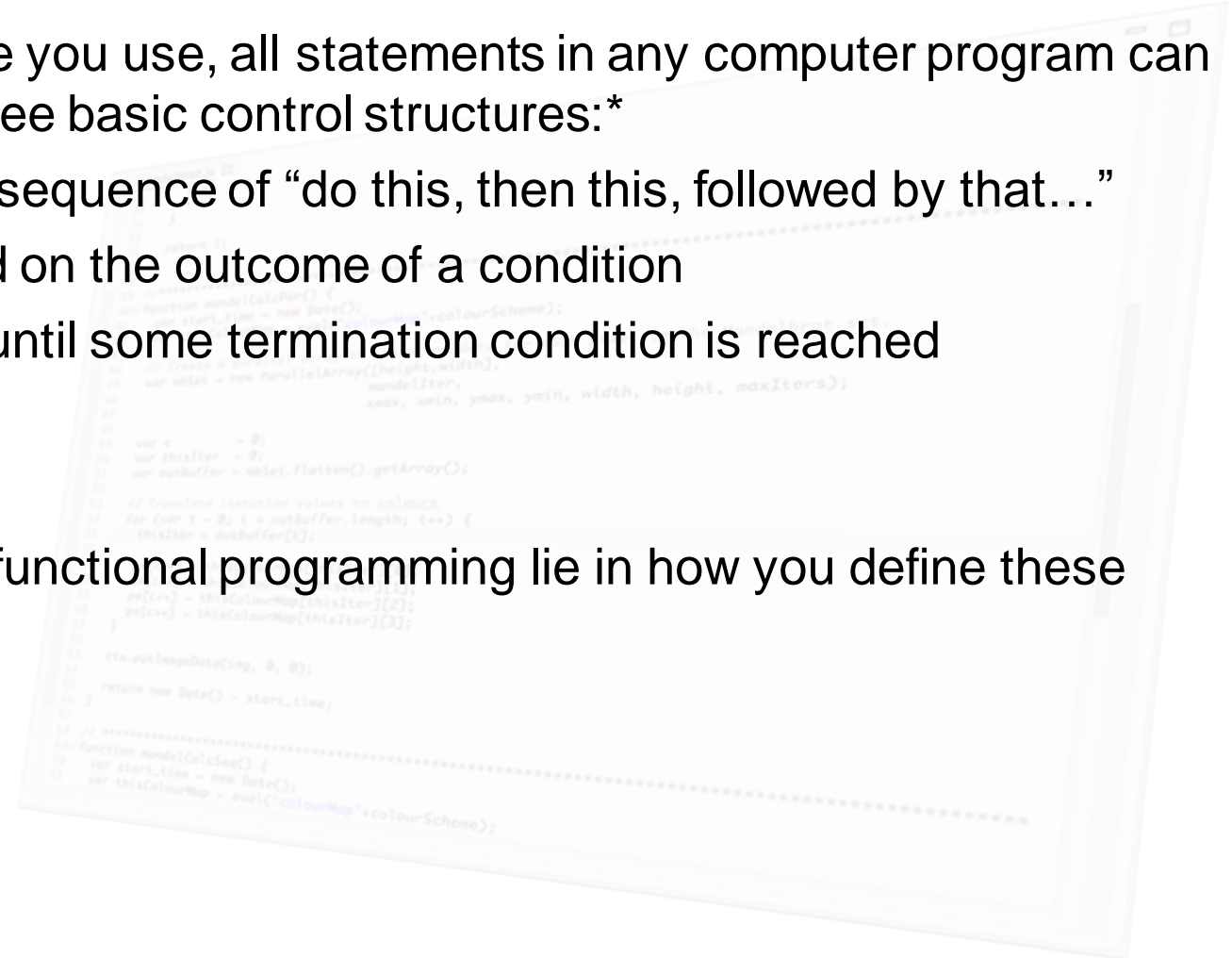


# Programming in General 3/3

No matter which programming style or language you use, all statements in any computer program can be grouped together in some combination of three basic control structures:\*

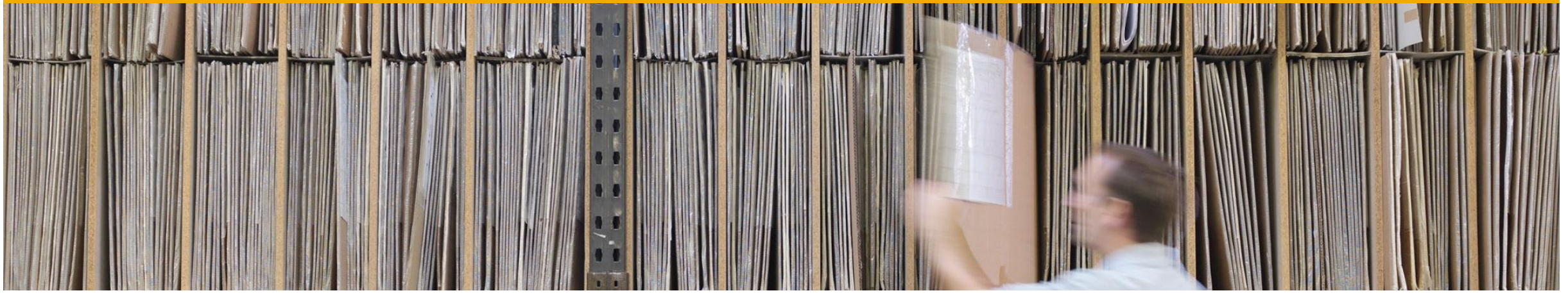
- Sequence**      The instruction flow is a simple sequence of “do this, then this, followed by that...”
- Selection**      Alters the instruction flow based on the outcome of a condition
- Iteration\*\***      Performs a task multiple times until some termination condition is reached

The major differences between imperative and functional programming lie in how you define these three control structures.



\* Known as the Böhm-Jacopini theorem

\*\* From the Latin “ite” meaning “go”



# The Differences Between Imperative and Functional Programming

# Imperative Programming: Do exactly this and do it now!

The “imperative programming” school of thought is by far the most widely used style of coding and is implicitly assumed to be the “normal” way to program a computer.

This style of programming arose from the computer architecture defined by John von Neumann in 1945 and is best suited to situations in which a known computation must be performed on a predictable dataset.

In this style of coding, the programmer is focused primarily on:

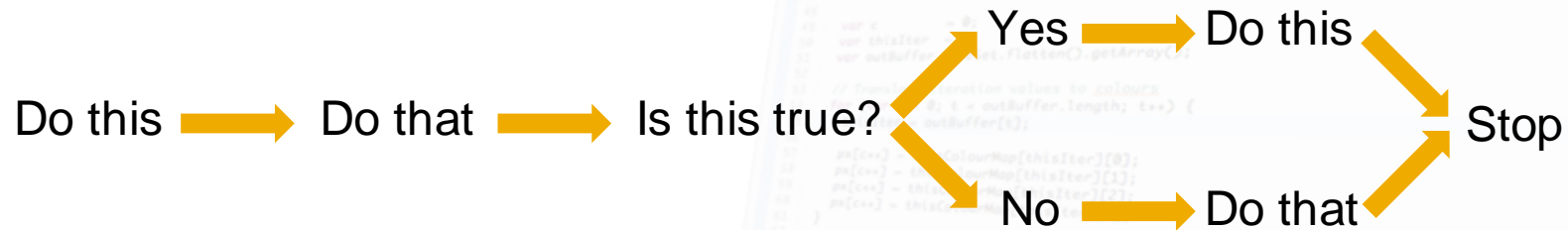
- Transforming the computer’s state (I.E. the data in memory) by means of a precisely defined set of instructions.
- Making explicit use of beneficial side-effects to achieve the desired result:
  - The use of global or shared memory as the repository for the evolving state of the data
  - Interacting with persistent storage (E.G a database)
  - Interacting with peripheral devices (printer, network, camera, accelerometer etc)



# Imperative Programming: Focusing on the “*when*”

In spite of the fact that “Imperative programming” is a broad term that encompasses many other styles of coding (E.G. procedural, modular, object oriented etc), the programmer’s job is essentially concerned with defining the explicit timeline of ***when*** statements should be executed. In other words:

*An imperative program is a set of instructions that define exactly ***when*** the computer’s state should be modified*



# Imperative Programming: Summary

Imperative programming languages give the programmer the necessary tools for defining the three fundamental control structures **explicitly**

**Sequence**      A set of instructions whose execution order is explicitly defined

**Selection**      Implemented using statements such as if or switch or case that modify the instruction flow based on the outcome of an explicitly defined condition

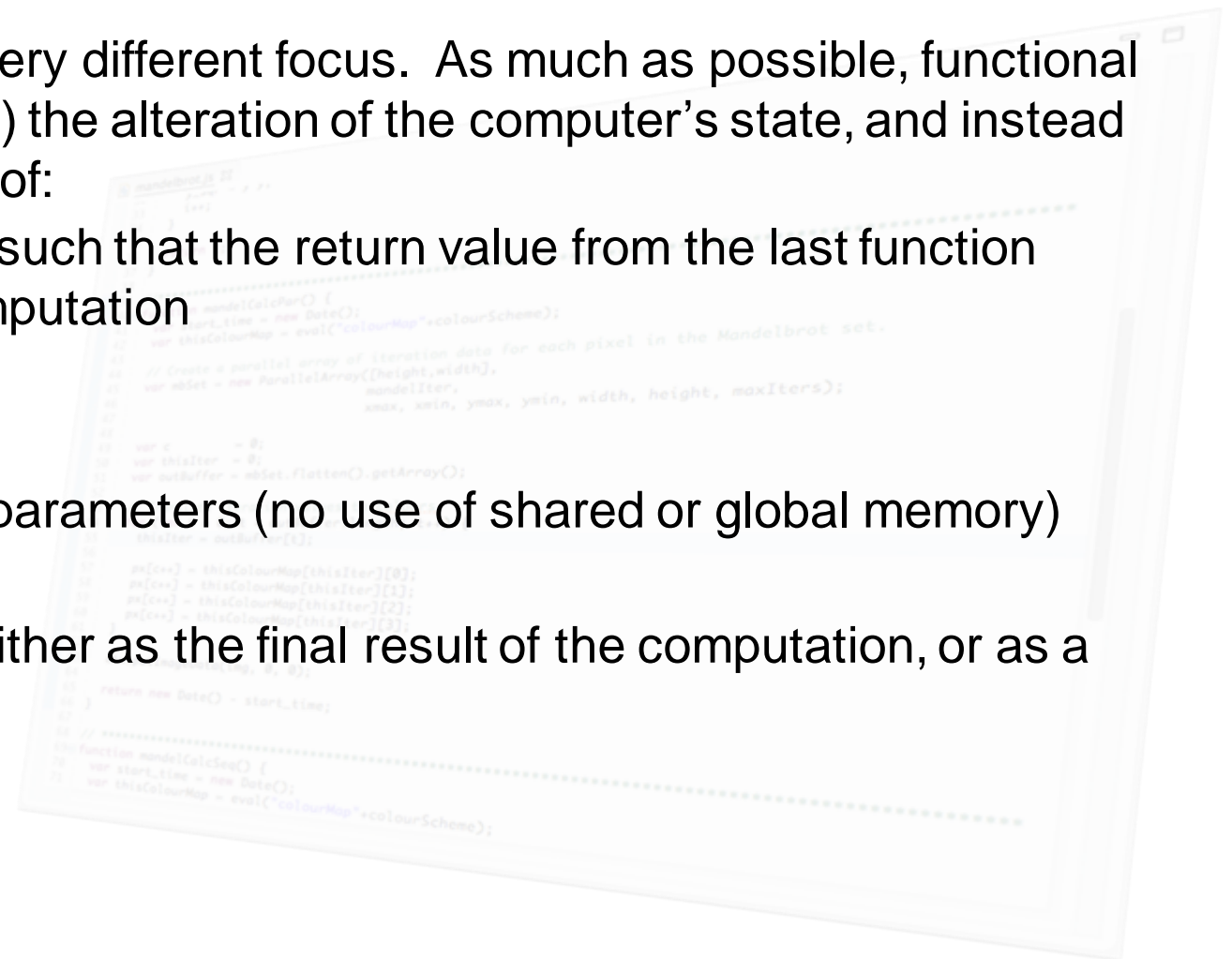
**Iteration**      Implemented using statements such as do/while or for loops

Since imperative programming is strongly focused on the timeline of **when** the computer's state should be modified, these languages necessarily must provide the programmer with keywords that explicitly control the instruction flow (such as if and case and do/while).

# Functional Programming: Focusing on the “How”

In functional programming however, there is a very different focus. As much as possible, functional programming languages avoid (or even prohibit) the alteration of the computer’s state, and instead define the required computation solely in terms of:

- The relationship between a set of functions, such that the return value from the last function corresponds to the desired output of the computation
- Each function:
  - Receives one or more parameters
  - Acts **only** upon the values received as parameters (no use of shared or global memory)
  - Returns a result to the calling function
- The result returned from a function is used either as the final result of the computation, or as a parameter value passed to another function.





# Functional Programming: Summary

Functional programming languages are still required to arrange their statements according to the three fundamental control structures seen before, but the tools they provide perform this task **implicitly**

**Sequence** A set of instructions whose execution order is explicitly defined

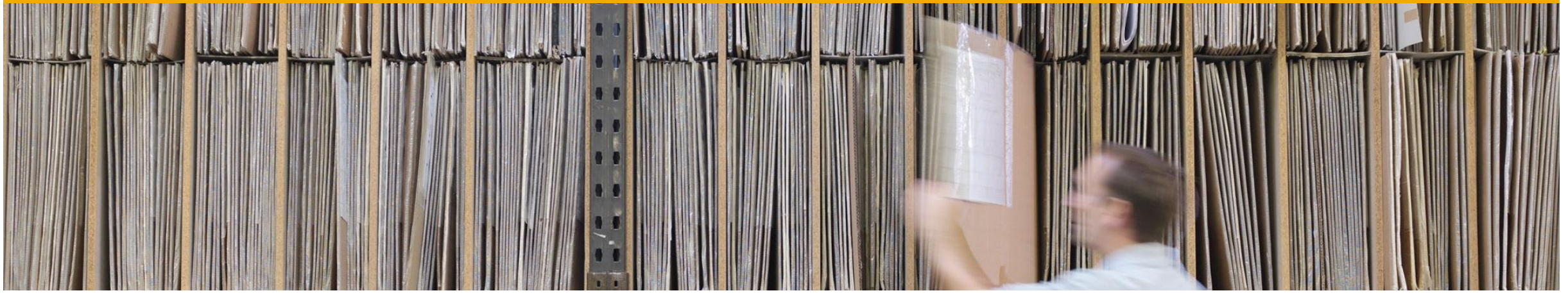
**Selection** Implemented by the language runtime using pattern matching, rather than by an explicitly coded condition such as `if` or `switch` or `case`.

**Iteration** Implemented by means of recursion.

Functional programming is strongly focused on creating a solution that avoids side-effects (the manipulation of global or shared memory is considered to be a side-effect).

Therefore, this style of programming focuses on the relationship between the input data and the output data, then defines a set of functions that perform the required transformation. This results in a greatly reduced focus on writing code to test explicit conditions or to repeat blocks of code some predefined number of times.

Instead, these aspects of modifying the instruction flow are delegated to the language runtime.



# Functional Programming

## A Simple Example

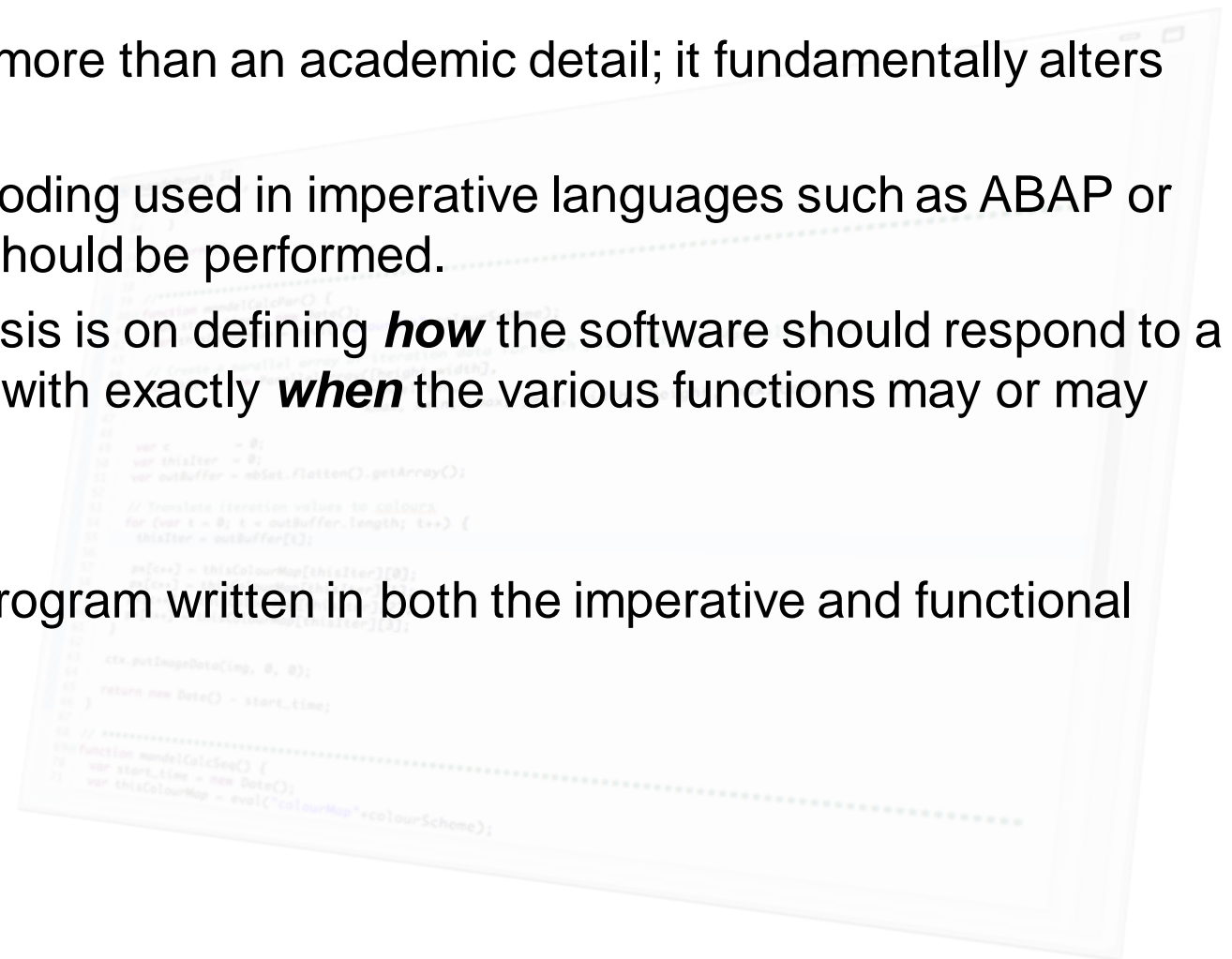
# Functional Programming: A Worked Example

This shift of emphasis from **when** to **how** is far more than an academic detail; it fundamentally alters the way you think about software design!

A programmer familiar with the architecture of coding used in imperative languages such as ABAP or Java is strongly focused on **when** instructions should be performed.

However in functional programming, the emphasis is on defining **how** the software should respond to a given pattern of data, and is far less concerned with exactly **when** the various functions may or may not be called.

This is best illustrated with a small JavaScript program written in both the imperative and functional styles.





# Calculating Fibonacci Numbers: Imperative Style 1/4

Here is a simple JavaScript function that calculates the  $n$ th number in the Fibonacci sequence (using the formula  $\text{fib}_n = \text{fib}_{n-2} + \text{fib}_{n-1}$ )

```
// Calculate nth Fibonacci number: imperative style
```

```
function fib_imperative(n) {
```

```
  var a = 0, b = 1, sum = 0;
```

```
  while (n>1) {
```

```
    sum = a + b;
```

```
    a = b;
```

```
    b = sum;
```

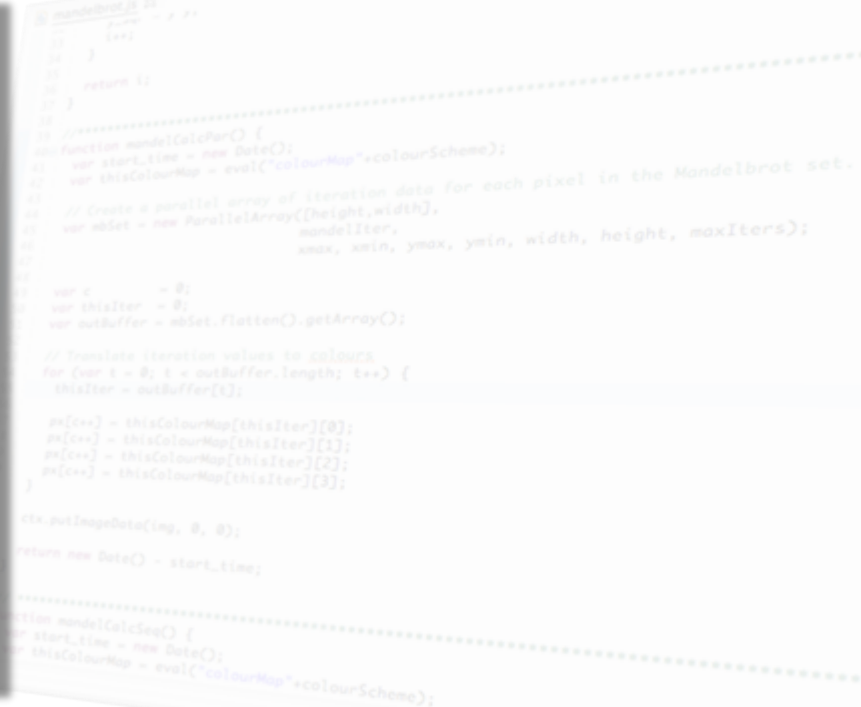
```
    n = n - 1;
```

```
  }
```

```
  return sum;
```

```
}
```

```
fib_imperative(8); // → 21
```



```
mandelbrot.js 22
// ...
23   }
24   }
25   return 1;
26 }
27 }
28
29 // ...
30 function mandelCalcPar() {
31   var start_time = new Date();
32   var thisColourMap = eval("colourMap"+colourScheme);
33
34   // Create a parallel array of iteration data for each pixel in the Mandelbrot set.
35   var mblt = new ParallelArray([height,width],
36     mandelIter,
37     xmax, xmin, ymax, ymin, width, height, maxIters);
38
39   var x = 0;
40   var thisIter = 0;
41   var outBuffer = mblt.flatten().getArray();
42
43   // Translate iteration values to colours
44   for (var i = 0; i < outBuffer.length; i++) {
45     thisIter = outBuffer[i];
46
47     px[i++] = thisColourMap[thisIter][0];
48     px[i++] = thisColourMap[thisIter][1];
49     px[i++] = thisColourMap[thisIter][2];
50     px[i++] = thisColourMap[thisIter][3];
51   }
52
53   ctx.putImageData(img, 0, 0);
54   return new Date() - start_time;
55 }
56
57 // ...
58 function mandelCalcSeq() {
59   var start_time = new Date();
60   var thisColourMap = eval("colourMap"+colourScheme);
61
62   // ...
63 }
```

# Calculating Fibonacci Numbers: Imperative Style 2/4

Here is a simple JavaScript function that calculates the  $n$ th number in the Fibonacci sequence (using the formula  $\text{fib}_n = \text{fib}_{n-2} + \text{fib}_{n-1}$ )

```
// Calculate nth Fibonacci number: imperative style
function fib_imperative(n) {
    var a = 0, b = 1, sum = 0;

    while (n > 1) {
        sum = a + b;
        a = b;
        b = sum;
        n = n - 1;
    }

    return sum;
}

fib_imperative(8); // → 21
```

Function `fib_imperative()` receives a single parameter `n` and uses this value to control a simple **while** loop.

Inside the loop, three variables (`a`, `b` and `sum`) are used to maintain the state of the calculation.

# Calculating Fibonacci Numbers: Imperative Style 3/4

Here is a simple JavaScript function that calculates the  $n$ th number in the Fibonacci sequence (using the formula  $\text{fib}_n = \text{fib}_{n-2} + \text{fib}_{n-1}$ )

```
// Calculate nth Fibonacci number: imperative style
function fib_imperative(n) {
  var a = 0, b = 1, sum = 0;

  while (n > 1) {
    sum = a + b;
    a = b;
    b = sum;
    n = n - 1;
  }

  return sum;
}

fib_imperative(8); // → 21
```

Function `fib_imperative()` receives a single parameter `n` and uses this value to control a simple **while** loop.

Inside the loop, three variables (`a`, `b` and `sum`) are used to maintain the state of the calculation.

The final value is accumulated in variable `sum` which then becomes the return value.

Notice that as the loop progresses, the value of variable `sum` continually changes. This is a classic example of the imperative coding style in which the computer's state evolves towards the desired result.



# Calculating Fibonacci Numbers: Imperative Style 4/4

Here is a simple JavaScript function that calculates the  $n$ th number in the Fibonacci sequence (using the formula  $\text{fib}_n = \text{fib}_{n-2} + \text{fib}_{n-1}$ )

```
// Calculate nth Fibonacci number: imperative style
function fib_imperative(n) {
  var a = 0, b = 1, sum = 0;

  while (n > 1) {
    sum = a + b;
    a = b;
    b = sum;
    n = n - 1;
  }

  return sum;
}

fib_imperative(8); // → 21
```

Function `fib_imperative()` receives a single parameter `n` and uses this value to control a simple **while** loop.

Inside the loop, three variables (`a`, `b` and `sum`) are used to maintain the state of the calculation.

The final value is accumulated in variable `sum` which then becomes the return value.

Notice that as the loop progresses, the value of variable `sum` continually changes. This is a classic example of the imperative coding style in which the computer's state evolves towards the desired result.

Because we are all familiar with the imperative coding style, this program is quite understandable.

Now let's see exactly the same functionality written in the functional style.

# Calculating Fibonacci Numbers: Functional Style

Not only is this program much more compact, but familiar statements such as `if` and `while` are missing. This solution will require some explanation...

```
// Calculate nth Fibonacci number: functional style
```

```
function fib_functional(n) {  
  return (function do_fib(a,b,n) {  
    return (n==1) ? b : do_fib(b,a+b,n-1);  
  })(0,1,n);  
}
```

```
fib_functional(8); // → 21
```

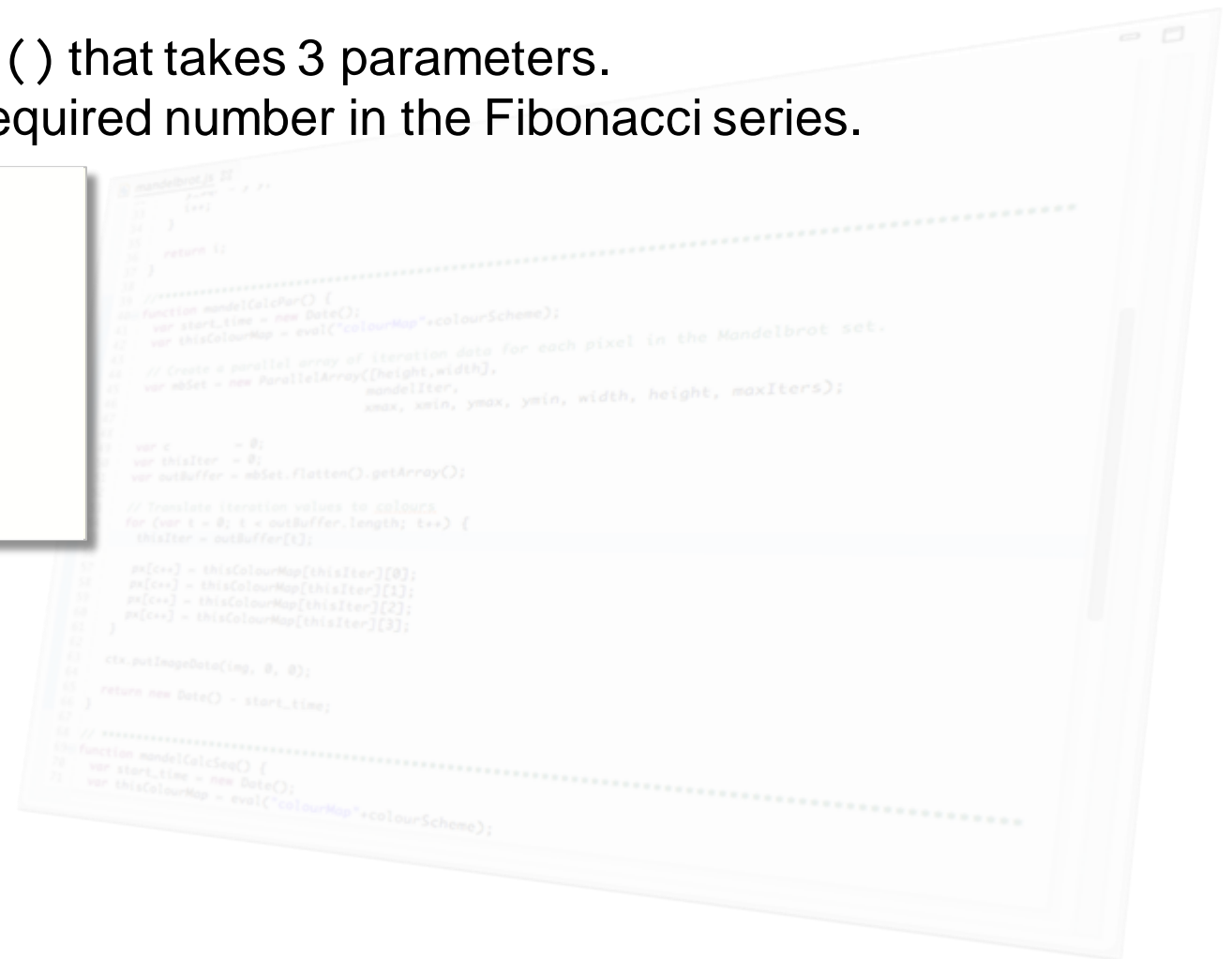


# Calculating Fibonacci Numbers: Explanation of Functional Style 1/5

At the centre, we have a function called `do_fib()` that takes 3 parameters. Parameter `a` is  $\text{fib}_{n-2}$ , `b` is  $\text{fib}_{n-1}$  and `n` is the required number in the Fibonacci series.

```
// Calculate nth Fibonacci number: functional style
function fib_functional(n) {
  return (function do_fib(a,b,n) {
    return (n==1) ? b : do_fib(b,a+b,n-1);
  })(0,1,n);
}

fib_functional(8); // → 21
```





# Calculating Fibonacci Numbers: Explanation of Functional Style 2/5

There are several important differences about the coding style here:

```
// Calculate nth Fibonacci number: functional style
function fib_functional(n) {
  return (function do_fib(a,b,n) {
    return (n==1) ? b : do_fib(b,a+b,n-1);
  })(0,1,n);
}

fib_functional(8); // → 21
```

1. The ternary operator is used to create an inline condition.

Had we used an explicit **if** statement, then this would draw your attention to the loop counter variable `n` and make you think this value should somehow be the focus of our attention; when in reality, it is merely a means to an end.

The use of the ternary operator ensures that the focus of attention remains on the function's return value - which will either be the value held in parameter `b`, or the result of a recursive call to `do_fib()`.

# Calculating Fibonacci Numbers: Explanation of Functional Style 3/5

There are several important differences about the coding style here:

```
// Calculate nth Fibonacci number: functional style
function fib_functional(n) {
  return (function do_fib(a,b,n) {
    return (n==1) ? b : do_fib(b,a+b,n-1);
  })(0,1,n);
}

fib_functional(8); // → 21
```

2. Notice that no internal variables are declared within function `do_fib()`.

The actual calculation of the next number in the Fibonacci sequence is performed at the time the parameters are passed to the recursive `do_fib()` call.

3. Use of the **while** keyword is no longer required because iteration is now controlled by recursion.

# Calculating Fibonacci Numbers: Explanation of Functional Style 4/5

There are several important differences about the coding style here:

```
// Calculate nth Fibonacci number: functional style
function fib_functional(n) {
  return (function do_fib(a,b,n) {
    return (n==1) ? b : do_fib(b,a+b,n-1);
  })(0,1,n);
}

fib_functional(8); // → 21
```

4. Function `do_fib()` is invoked automatically by enclosing the function definition within parentheses and then using the invocation operator `()`.
5. The invocation operator contains the 3 required parameters: 0 and 1 (the first two numbers in the Fibonacci sequence) and *n* (the *n*th number in the Fibonacci sequence).

# Calculating Fibonacci Numbers: Explanation of Functional Style 5/5

There are several important differences about the coding style here:

```
// Calculate nth Fibonacci number: functional style
function fib_functional(n) {
  return (function do_fib(a,b,n) {
    return (n==1) ? b : do_fib(b,a+b,n-1);
  })(0,1,n);
}

fib_functional(8); // → 21
```

6. The return value from the call to `do_fib()` becomes the return value of the enclosing function, here called `fib_functional()`
7. Function `fib_functional()` is a wrapper function that receives a single parameter value and then acts as a container for the recursive calls to function `do_fib()`.





# Functional Programming

## Side-effects Are Bad!

# Functional Programming – Avoid Side Effects!

When a function alters data outside its own scope, this is called a side-effect. This is generally bad because unexpected alterations to global data can lead to erroneous program behaviour that is very hard to debug.

```
var num = 1;    // Global variable

// A strange way to do addition...
function add_with_side_effects(a) {
    return a + num++;
}
```

# Functional Programming – Avoid Side Effects!

When a function alters data outside its own scope, this is called a side-effect. This is generally bad because unexpected alterations to global data can lead to erroneous program behaviour that is very hard to debug.

```
var num = 1;    // Global variable
```

```
// A strange way to do addition...
```

```
function add_with_side_effects(a) {  
    return a + num++;  
}
```

```
// Each time you call this function, it not only returns a result, but also increments a  
// global variable. This side-effect creates the problem that the next time you call the  
// same function with the same parameter, it will behave differently.  
// This makes the function unreliable and is therefore bad program design
```

```
add_with_side_effects(1);    // → 2, but 'num' now equals 2
```

```
add_with_side_effects(1);    // → 3 (Uh!?) and 'num' now equals 3
```

# Functional Programming – Referential Transparency 1/2

Referential transparency means that a function call can safely be replaced by the function's return value. Here's an example of a function that does not cause side-effects, but lacks referential transparency...

```
var num = 1;    // Global variable

// Another strange way to do addition...
function bad_add(a) {
    return a + num;
}

// Each time function bad_add() is called with the same parameter, it could potentially
// return a different result because its behaviour depends on a value that was not supplied
// as parameter. Again, this is bad design because it makes the function unreliable
var ans1 = bad_add(3); // → 4
```



# Functional Programming – Referential Transparency 1/2

Referential transparency means that a function call can safely be replaced by the function's return value. Here's an example of a function that does not cause side-effects, but lacks referential transparency...

```
var num = 1;    // Global variable

// Another strange way to do addition...
function bad_add(a) {
    return a + num;
}

// Each time function bad_add() is called with the same parameter, it could potentially
// return a different result because its behaviour depends on a value that was not supplied
// as parameter. Again, this is bad design because it makes the function unreliable
var ans1 = bad_add(3); // → 4

num = 4;        // Somewhere in the coding, the global variable is then changed...

var ans2 = bad_add(3); // → 7 (Uh!? Last time I called it, it returned 4)
```

# Functional Programming – Referential Transparency 2/2

This function does not create any side-effects and also has referential transparency. Any function that has referential transparency is said to be *idempotent*.

```
// Add up the supplied numbers
function good_add(a,b) {
  return a + b;
}
```

```
// Referential transparency means that no matter how many times function good_add() is
// called, as long as we pass the same parameters, we'll always get back the same result
var ans1 = good_add(2,3);    // → 5
var ans2 = good_add(2,3);    // → 5
```

# Functional Programming – Referential Transparency 2/2

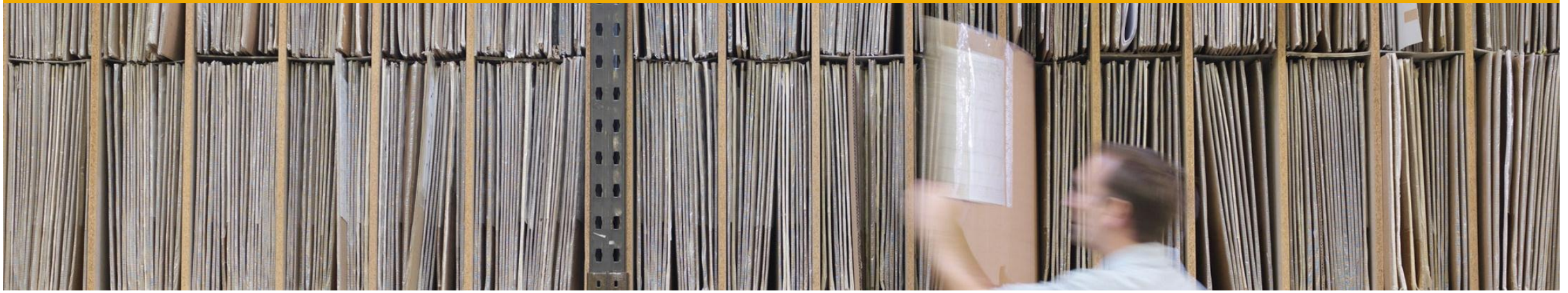
This function does not create any side-effects and also has referential transparency. Any function that has referential transparency is said to be ***idempotent***.

```
// Add up the supplied numbers
function good_add(a,b) {
  return a + b;
}

// Referential transparency means that no matter how many times function good_add() is
// called, as long as we pass the same parameters, we'll always get back the same result
var ans1 = good_add(2,3);    // → 5
var ans2 = good_add(2,3);    // → 5
```

To achieve referential transparency (or idempotency), always ensure that a function:

- a) returns a value derived **only** from its parameters
- b) never relies on the value of global variables
- c) does not cause side-effects by modifying data outside its own scope



# Functional Programming

## A Simple Example From SAPUI5



# Functional Programming – Example from SAPUI5 1/6

Handling user events is an example of where we need to focus the **how** and not the **when**:

- Define a function to handle a particular pattern of data (E.G. a button push event)

```
// Define an event handler function for a button UI element
var evtHandler = function() { alert("B A N G !"); }
var btn = new sap.ui.commons.Button({text:'Do not push this button'});
```



Do not push this button

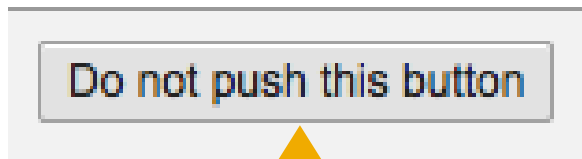
# Functional Programming – Example from SAPUI5 2/6

Handling user events is an example of where we need to focus the **how** and not the **when**:

- Define a function to handle a particular pattern of data (E.G. a button push event)
- Attach the event handler function to the appropriate event of the UI element

```
// Define an event handler function for a button UI element
var evtHandler = function() { alert("B A N G !"); }
var btn = new sap.ui.commons.Button({text:'Do not push this button'});

// Attach event handler function to the "press" event of the button UI element
btn.attachPress(evtHandler);
```



Attach an event  
handler function



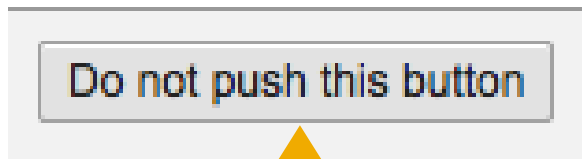
# Functional Programming – Example from SAPUI5 3/6

Handling user events is an example of where we need to focus the **how** and not the **when**:

- Define a function to handle a particular pattern of data (E.G. a button push event)
- Attach the event handler function to the appropriate event of the UI element

```
// Define an event handler function for a button UI element
var evtHandler = function() { alert("B A N G !"); }
var btn = new sap.ui.commons.Button({text:'Do not push this button'});

// Attach event handler function to the "press" event of the button UI element
btn.attachPress(evtHandler);
```



Attach an event  
handler function



Wait for an unknown  
period of time

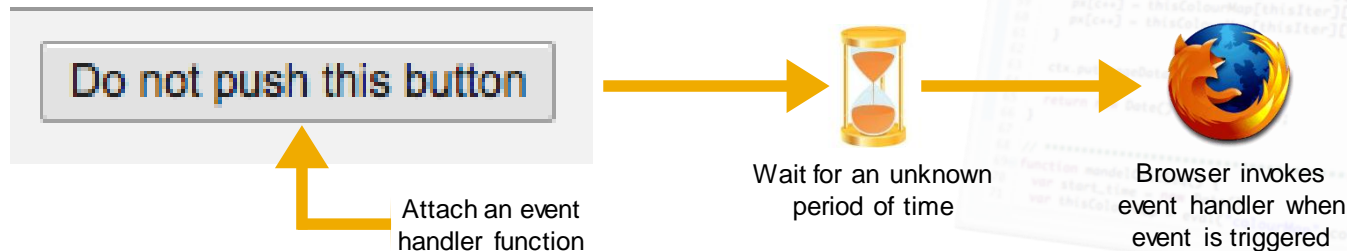
# Functional Programming – Example from SAPUI5 4/6

Handling user events is an example of where we need to focus the **how** and not the **when**:

- Define a function to handle a particular pattern of data (E.G. a button push event)
- Attach the event handler function to the appropriate event of the UI element

```
// Define an event handler function for a button UI element
var evtHandler = function() { alert("B A N G !"); }
var btn = new sap.ui.commons.Button({text:'Do not push this button'});

// Attach event handler function to the "press" event of the button UI element
btn.attachPress(evtHandler);
```



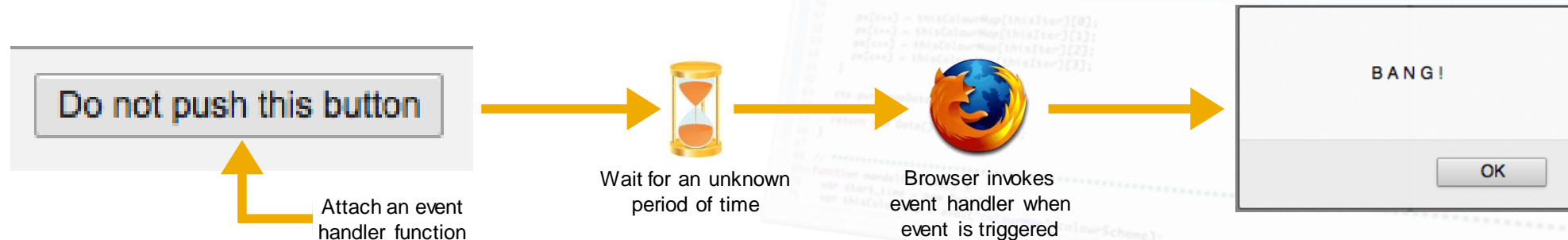
# Functional Programming – Example from SAPUI5 5/6

Handling user events is an example of where we need to focus the **how** and not the **when**:

- Define a function to handle a particular pattern of data (E.G. a button push event)
- Attach the event handler function to the appropriate event of the UI element

```
// Define an event handler function for a button UI element
var evtHandler = function() { alert("B A N G !"); }
var btn = new sap.ui.commons.Button({text:'Do not push this button'});

// Attach event handler function to the "press" event of the button UI element
btn.attachPress(evtHandler);
```





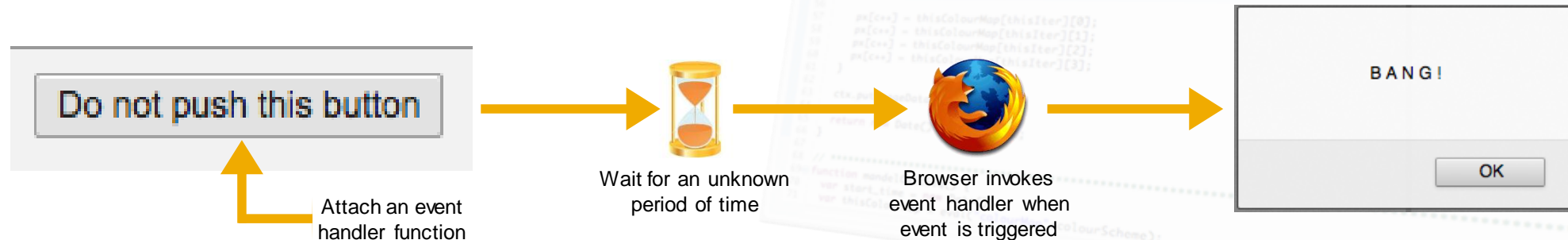
# Functional Programming – Example from SAPUI5 6/6

Handling user events is an example of where we need to focus the **how** and not the **when**:

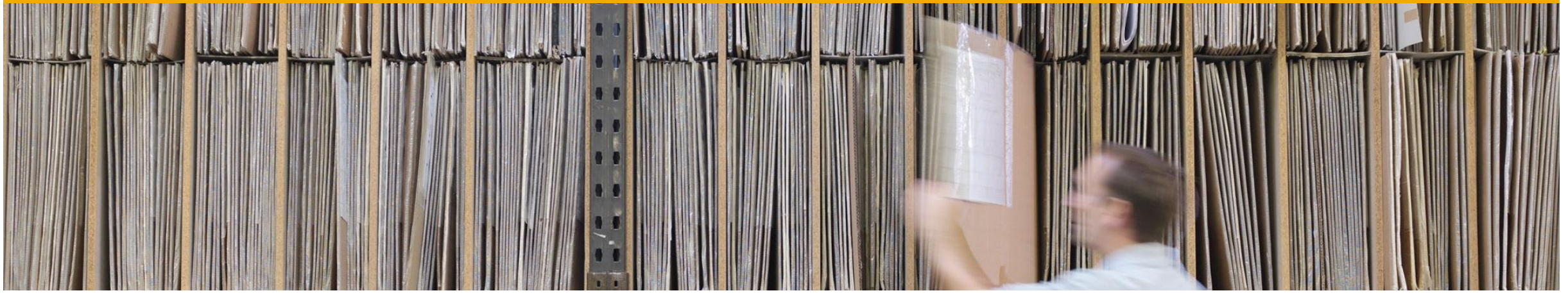
- Define a function to handle a particular pattern of data (E.G. a button push event)
- Attach the event handler function to the appropriate event of the UI element

```
// Define an event handler function for a button UI element
var evtHandler = function() { alert("B A N G !"); }
var btn = new sap.ui.commons.Button({text:'Do not push this button'});

// Attach event handler function to the "press" event of the button UI element
btn.attachPress(evtHandler);
```



This style of programming focuses entirely on **how** the system should respond when a pattern of data is received (E.G. a UI event). **When** we respond is not important because the button might never be pushed!



# Functional Programming

## Summary

# Functional Programming – Summary 1/2

Functional programming is style of coding that borrows heavily from a branch of mathematics known as Lambda Calculus.

The functional programming style imposes the following constraints:

- Functions interact with each other **only** by means of parameters and return values.
- The use of internal variables to maintain program state is avoided (or in some languages prohibited).
- Side-effects are not permitted. Most functional programming languages do not have the concept of shared or global memory.
- Looping is handled by means of recursion, rather than explicit loop constructs such as `do...while` or `for`.
- The exact order in which functions are invoked is delegated to the language runtime on the basis of the data available at execution time.

The appeal of the functional programming style is that if you strictly follow its design philosophy, it allows you to write coding that is provably correct (as apposed to hopefully correct).

This feature has always had great appeal in academic circles, but with the increasing popularity of multi-core processors and the need to analyze ever increasing quantities of data, the use of the functional programming style is steadily increasing in the world of business programming.

# Functional Programming – Summary 2/2

Different functional programming languages implement these principles to varying degrees.

Those languages that fully implement all the principles are said to be “purely functional” languages (such as Haskell or Miranda), with the rest being referred to as “impurely functional”.

Impure functional languages are generally easier to code in because you do not need to resort to such deep levels of mathematical abstraction. Examples of impure functional languages include:

- Erlang
- F#
- Lisp
- R (Used internally with SAP's HANA database)

## **CAUTION!**

JavaScript is a multi-paradigm language! This means that both the imperative and functional styles of coding can co-exist within the same application.

Care must be taken when designing JavaScript applications to ensure that confusion is not introduced into the architecture by a jumbled use of these different paradigms.

