

COMP6080

Transpilation

Presented by
Nic Barker

Canva

Definition

What is
transpilation?

Definition

Compilation

source code \rightarrow machine code

Transpilation

source code \rightarrow source code

Definition

Compilation

C++ \rightarrow Machine code

Transpilation

C++ \rightarrow Javascript

The why

Transpilation

Why do we need it?

What are the use cases?

The why

Other language -> Javascript

Javascript is the only natively supported language in the browser**. This means that by default if you want to use a different language to write your frontend web apps, you need to transpile to JS.

**yes there's wasm now, it's still not totally well supported /
tooled

Source (C++) -> Transpiled (Javascript)

```
C test.c ×
C test.c
1 #include <stdio.h>
2
3 int main() {
4     printf("hello, world!\n");
5     return 0;
6 }
7

JS a.out.js ×
JS a.out.js > ...
5918     buffer.length = 0;
5919 } else {
5920     buffer.push(curr);
5921 }
5922 },varargs:undefined,get:function() {
5923     assert(SYSCALLS.varargs != undefined);
5924     SYSCALLS.varargs += 4;
5925     var ret = HEAP32[(((SYSCALLS.varargs)-(4))>>2)];
5926     return ret;
5927 },getStr:function(ptr) {
5928     var ret = UTF8ToString(ptr);
5929     return ret;
5930 },get64:function(low, high) {
5931     if (low >= 0) assert(high === 0);
5932     else assert(high === -1);
5933     return low;
5934 }};
5935 function _fd_write(fd, iov, iovcnt, pnum) {
5936     // hack to support printf in SYSCALLS_REQUIRE_FILESYSTEM=0
5937     var num = 0;
5938     for (var i = 0; i < iovcnt; i++) {
5939         var ptr = HEAP32[(((iov)+(i*8))>>2)];
5940         var len = HEAP32[(((iov)+(i*8 + 4))>>2)];
5941         for (var j = 0; j < len; j++) {
5942             SYSCALLS.printChar(fd, HEAPU8[ptr+j]);
5943         }
5944         num += len;
5945     }
5946     HEAP32[((pnum)>>2)]=num;
5947     return 0;
5948 }
5949
5950 function _setTempRet0($i) {
5951     setTempRet0(($i) | 0);
5952 }
5953 var ASSERTIONS = true;
5954
5955
5956
5957 /** @type {function(string, boolean=, number=)} */
5958 function intArrayFromString(stringy, dontAddNull, length) {
5959     var len = length > 0 ? length : lengthBytesUTF8(stringy)+1;
5960     var u8array = new Array(len);
5961     var numBytesWritten = stringToUTF8Array(stringy, u8array, 0, u8array.length);
5962     if (dontAddNull) u8array.length = numBytesWritten;
5963     return u8array;
5964 }
5965
5966 function intArrayToString(array) {
5967     return stringFromUTF8Array(array, 0, array.length);
5968 }
```

The why

New Javascript -> Old Javascript

New features become available in Javascript that aren't supported in old browsers. We can **transpile** our new Javascript (e.g. **native js Class**) to compatible older APIs, while still using new features when we write our code.



Source

->

Transpiled (no class support)

```
1 class Test {  
2   classMethod() {  
3     console.log('test');  
4   }  
5 }
```

<

```
1 "use strict";  
2  
3 function _instanceof(left, right) { if (right != null && typeof  
Symbol !== "undefined" && right[Symbol.hasInstance]) { return  
!!right[Symbol.hasInstance](left); } else { return left instanceof  
right; } }  
4  
5 function _classCallCheck(instance, Constructor) { if  
(!_instanceof(instance, Constructor)) { throw new TypeError("Cannot  
call a class as a function"); } }  
6  
7 function _defineProperties(target, props) { for (var i = 0; i <  
props.length; i++) { var descriptor = props[i];  
descriptor.enumerable = descriptor.enumerable || false;  
descriptor.configurable = true; if ("value" in descriptor)  
descriptor.writable = true; Object.defineProperty(target,  
descriptor.key, descriptor); } }  
8  
9 function _createClass(Constructor, protoProps, staticProps) { if  
(protoProps) _defineProperties(Constructor.prototype, protoProps);  
if (staticProps) _defineProperties(Constructor, staticProps);  
return Constructor; }  
10  
11 var Test = /*#__PURE__*/function () {  
12   function Test() {  
13     _classCallCheck(this, Test);  
14   }  
15  
16   _createClass(Test, [{  
17     key: "classMethod",  
18     value: function classMethod() {  
19       console.log('test');  
20     }  
21   }]);  
22  
23   return Test;  
24 }();
```

The why

You can try compiling new -> old
Javascript at

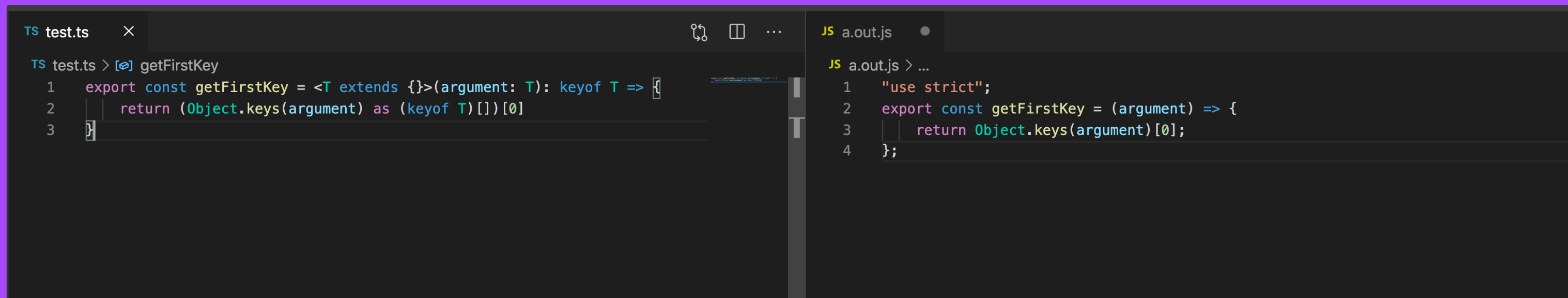
<https://babeljs.io/repl>

The why

Almost Javascript -> Javascript

Flavours of Javascript have been created for syntactic sugar or **type safety**. For example, **Coffeescript** or **Typescript**. These are almost JS (much closer than C++) but still need to be **transpiled** to vanilla JS to run in the browser.

Typescript → Javascript



```
TS test.ts > [0] getFirstKey
1  export const getFirstKey = <T extends {}>(argument: T): keyof T => {
2    return (Object.keys(argument) as (keyof T)[])[0]
3  }
```

```
JS a.out.js > ...
1  "use strict";
2  export const getFirstKey = (argument) => {
3    return Object.keys(argument)[0];
4  };
```

The why

Try the typescript compiler:

`typescriptlang.org/play`

The why

JSX transformation

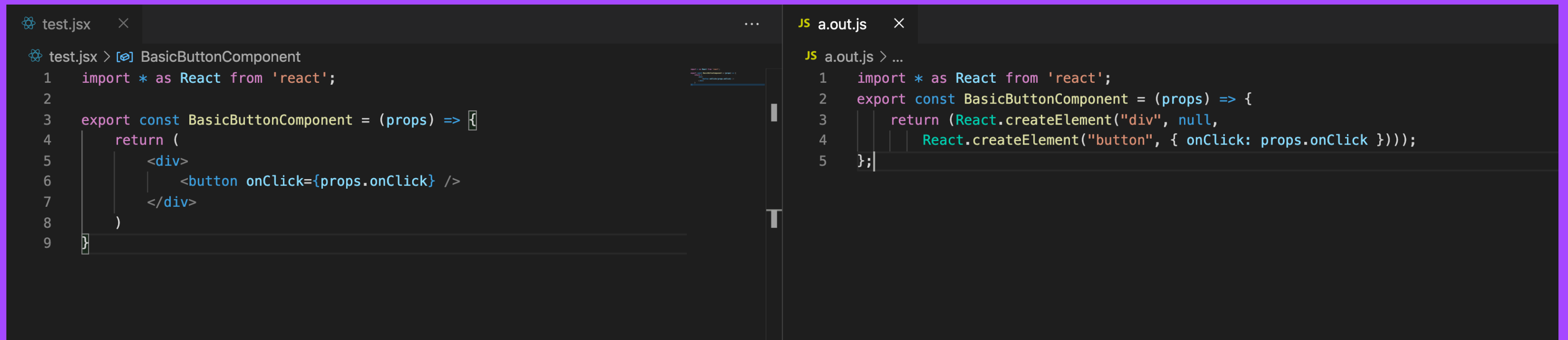
React is just Javascript. It doesn't use any string or HTML templating.

Under the hood when you write something like
component = <div><button /></div>

what you're actually writing is:

```
component = React.createElement("div", null,  
  React.createElement("button", null));
```

JSX → Javascript



```
test.jsx > [🔗] BasicButtonComponent
1  import * as React from 'react';
2
3  export const BasicButtonComponent = (props) => {
4    return (
5      <div>
6        <button onClick={props.onClick} />
7      </div>
8    )
9  }
```

```
JS a.out.js > ...
JS a.out.js > ...
1  import * as React from 'react';
2  export const BasicButtonComponent = (props) => {
3    return (React.createElement("div", null,
4      React.createElement("button", { onClick: props.onClick })));
5  };
```

The why

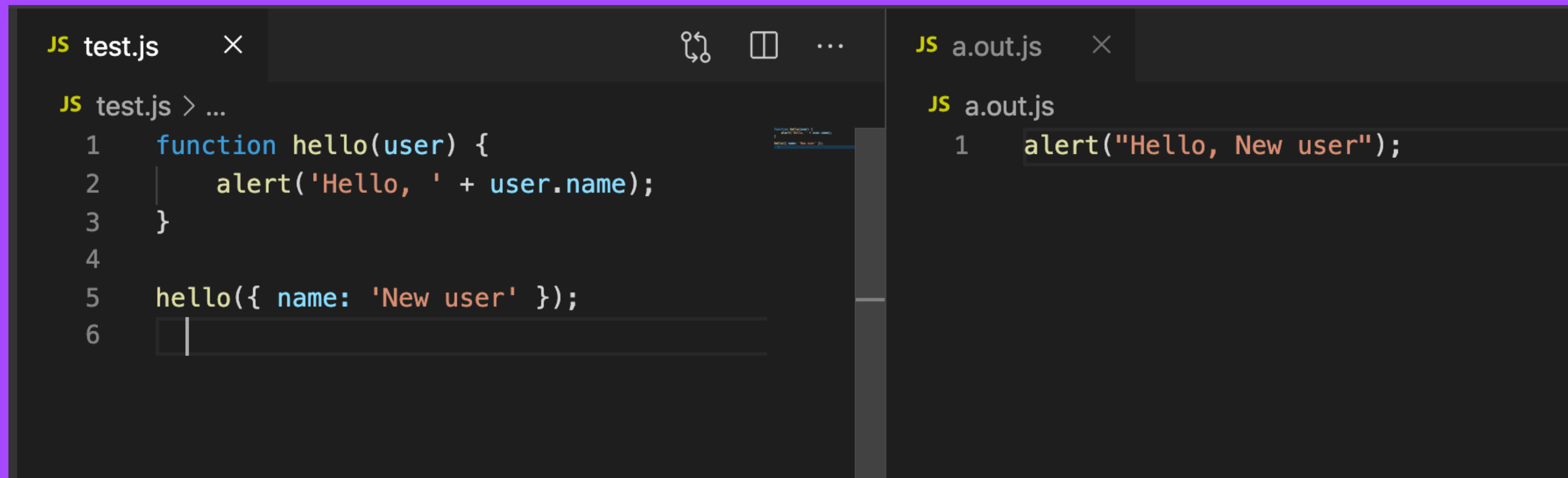
Minification + Obfuscation

Javascript **Minification** and / or **Obfuscation** are both technically types of **transpilation**.

Minification / Obfuscation takes our existing code and condenses it, removing whitespace, removing comments, shortening variable names and in some cases even automatically refactoring control flow to save network transfer when downloading scripts.

This process also makes it more difficult for competitors or malicious actors to reverse engineer our source code.

Min / Obfs example: Google "Closure Compiler" in advanced mode can determine that the function **hello** and the **user** object are only used once, and removes them completely



The image shows a side-by-side comparison of JavaScript code before and after minification. The left pane, titled 'test.js', contains the original code: a function 'hello' that takes a 'user' object and calls 'alert' with a message, followed by a call to 'hello' with a specific user object. The right pane, titled 'a.out.js', shows the result after minification: the function definition and the variable 'user' have been removed, leaving only the 'alert' call with the stringified user data.

```
JS test.js ×
JS test.js > ...
1 function hello(user) {
2   alert('Hello, ' + user.name);
3 }
4
5 hello({ name: 'New user' });
6

JS a.out.js ×
JS a.out.js
1 alert("Hello, New user");
```

over a large application, this can save a lot of bytes

The why

Try it out:

`https://closure-compiler.appspot.com/home`

The how

How does **transpiling** generally work?

Before we deploy our Javascript code to production, we run a **build** step using tooling like **webpack** or **rollup** or **gulp**.

The how

Tools like webpack are usually a chain of plugins that handle lots of different file types and transpilation steps. In webpack they are called **loaders** but have different names ("plugins", etc) in other tools.

The how

Caveats

Transpiling isn't free. It takes time and will slow down your development / deployment experience if you aren't careful.

Heavy transpilation means the code executing in the browser may be difficult to debug. You will need good **source map** support to help you understand how the transpiled code maps back to the source code.