

Reading Sample

This chapter explores clean code best practices for two core components of your SAPUI5 code: variables and literals. You'll learn how to declare variables using the var, let, and const keywords, and you'll walk through the literal types for numbers, strings, arrays, objects, and more.

-  ["Variables and Literals"](#)
-  [Contents](#)
-  [Index](#)
-  [The Authors](#)

Daniel Bertolozzi, Arnaud Buchholz, Klaus Haeuptle, Rodrigo Jordão, Christian Lehmann, and Narendran Natarajan Vaithianathan

Clean SAPUI5: A Style Guide for Developers

530 pages, 2022, \$79.95

ISBN 978-1-4932-2228-5



www.sap-press.com/5479

Chapter 7

Variables and Literals

This chapter delves into how to declare and use variables and literals. Along the way, you'll learn how scoping rules work, how to use destructuring, and how to implement the spread syntax.

7

Variables, along with parameters, store values and references to the data used in your program. For any given piece of code, the number and complexity of the variables *in scope* affects how easy understanding and changing the code can be. In this chapter, we'll explore different ways to declare, initialize, and use variables. You'll also learn how scoping rules work for different variable declaration types and learn how to use literal values in code. We'll also talk about destructuring and the spread syntax when exploring arrays and object literals.

For a related discussion focused on functions and parameters, refer to Chapter 5.

7.1 Variables

In JavaScript, variables can be declared with the keywords `var`, `let`, and `const`.

Variables declared with `var` are *hoisted* to the start of the function (or global scope) in which they are declared. However, only the variable declaration is hoisted, not its initialization. Thus, a variable declared with `var` can be accessed before it's declared, but its value will be `undefined` until the point in the code where the variable definition is initialized, as shown in Listing 7.1.

```
function hoisting() {  
  console.log(message); // outputs "undefined"  
  var message = 'Hello World!';  
  console.log(message); // outputs "Hello World!"  
}
```

Listing 7.1 Hoisted `var` Accessed before Being Declared

A variable can even be initialized and used before its declaration, as shown in Listing 7.2, where the `message` variable is declared at the end of the scope of the `hoisting` function, but the variable is initialized and used before that declaration.

```
function hoisting() {
  console.log(message); // outputs "undefined"
  message = 'Hello World!';
  console.log(message); // outputs "Hello World!"

  var message;
}
```

Listing 7.2 Using a Variable before Its Declaration

Variables declared with `var` can also be declared multiple times in the same function scope, and they'll all refer to the same variable. As shown in Listing 7.3, the `message` variable is declared twice: once at the beginning of the redeclaration function and another time inside the scope of the `if` statement in the function. Both declarations actually refer to the same variable since the scope of `var` is the scope of the whole function.

```
function redeclaration() {
  var message = 'Hello World!';
  console.log(message); // outputs "Hello World!"

  if (true) {
    var message = 'Howdy World!';
    console.log(message); // outputs "Howdy World!"
  }

  console.log(message); // outputs "Howdy World!"
}
```

Listing 7.3 Variable Redefinition

ESLint has rules that deal with all these issues with `var`. For example, the `no-use-before-define` rule can check for variable usage before variable definition and can be configured to check for function, class, and variable declarations. However, you should only use this rule for variables; the `no-redeclare` rule can check for multiple declarations of the same variable.

Do Not Use `var`

The `var` keyword is the oldest of the three variable declaration keywords and is retained in the language for backwards compatibility. Its behavior can sometimes be surprising and lead to hard-to-understand code, and thus, its usage in modern JavaScript codebases is discouraged. Do not use the `var` keyword; instead, try to refactor this declaration to `let` or `const` when you find it. The `no-var` rule in ESLint can help enforce this guideline.

The antidote to `var` is declaring variables with `let` or `const`. Both these keywords for variable declaration are sane and don't allow the kind of confusing code that's possible with `var`. Using a variable declared with `let` or `const` before its declaration raises a `ReferenceError`. Redeclaring a variable in the same scope raises a `SyntaxError`.

The scope for variables declared with `let` or `const` is not the whole function scope, as with `var`, but instead, the scope is code between the curly brackets `{` and `}`, also known as the *block scope*. Redeclaring a variable is possible in an inner scope within a function, but the variable declared in the inner scope is a new variable that shadows the definition of the outer scope variable. Let's rewrite the example shown in Listing 7.3 using `let` and see what happens, as shown in Listing 7.4.

```
function shadowing() {
  let message = 'Hello World!';
  console.log(message); // outputs "Hello World!"

  if (true) {
    let message = 'Howdy World!';
    console.log(message); // outputs "Howdy World!"
  }

  console.log(message); // outputs "Hello World!"
}
```

Listing 7.4 Variable Shadowing with `let`

The `message` variable declared in the outer scope is distinct from the one in the inner scope. This distinction is made obvious by the last line of the function, which outputs the value of the outer scope `message` after the inner scope (the scope within the `if` statement) has ended. Shadowing variable definitions in this way can still be confusing and should be avoided.

Declare Variables Close to Their First Use

Many legacy coding styles for languages like C or ABAP advocate for the use of a definition block at the beginning of functions or methods where all local variables are declared and optionally initialized. This approach, however, makes code harder to read and change. Declare and initialize variables as close as possible to where they are first used, thus minimizing their scope and increasing the understandability of the code. This approach also makes it easier to change and refactor code by extracting new methods or functions, for example.

The difference between `let` and `const` is that `let` allows the variable to be reassigned, whereas `const` does not. Note that `const` still allows a variable to be mutated (if it's not an immutable value type), so in that sense, a variable declared with `const` is not a true

constant. Using `let`, the variable `message` can be reassigned, as shown in Listing 7.5. If `message` was declared with `const`, the reassignment in `message = 'Goodbye'` would raise a `TypeError`.

```
function reassignment() {
  let message = 'Hello';
  console.log(message);
  message = 'Goodbye';
  console.log(message);
}
```

Listing 7.5 Variable Reassignment

Use `const` by Default

Clean SAPUI5 guidelines encourage the creation of small functions with simple and understandable code, usually with a small number of variable declarations. These variables will likely be assigned only once during initialization, where `const` can be used. Using `const` also makes the code easier to reason about, since you know that this variable will not be reassigned. Use `const` by default in all variable declarations. Only use `let` if the variable will be reassigned.

The `let` keyword can be used for loop indexes, as shown in Listing 7.6. Remember that the construct `++i` is equivalent to `i = i + 1`, which is a reassignment of `i`.

```
const numbers = [1, 2, 3, 4];

for (let i = 0; i < numbers.length; ++i) {
  const number = numbers[i];
  console.log(`The number in index ${i} is ${number}`);
}
```

Listing 7.6 Loop Index Declared with `let`

However, if you don't use the index directly, you can use the `for...of` construct and use `const` to iterate over the elements of an array (or any iterable), as shown in Listing 7.7.

```
const numbers = [1, 2, 3, 4];

for (const number of numbers) {
  console.log(number);
}
```

Listing 7.7 `for...of` Loop Using `const`

Multiple variables can also be declared with a single declaration statement, separated by a comma (,), as shown in Listing 7.8, which is a rewriting of the `copySelectedProduct` function we discussed earlier in Chapter 5, Section 5.11.12 (functions) and Chapter 6, Section 6.9.2 (naming). However, declaring multiple variables in a single statement in this manner can make the code harder to read and maintain and thus should be avoided. Strive to only declare one variable in each declaration statement.

```
async copySelectedProduct() {
  const
    sourceProductId = this.getSelectedProductId(),
    targetProductId = await this.copyProduct(sourceProductId);
  this.navToProductEdit(targetProductId);
},
```

Listing 7.8 Multiple Variables in One Declaration

The `this` variable is a special variable associated with the current execution context. For more on the `this` variable, take a look at Chapter 5.

7.2 Literals

Literals are hard-coded values in the code, like the number `42` or the string `"Hello"`. Literals can be strings, numbers, regular expressions, Booleans, and any other type that can be written as a value, even objects and arrays.

Usually, when you have literals in your code, consider whether moving that literal to a constant, an enum element, or a single shared definition that will name the literal makes sense to facilitate understanding and later changes in the code.

When applied to numbers, for example, literals in the middle of the code are usually referred to as *magic numbers*. Let's look at the code shown in Listing 7.9 for an example of the use of a magic number.

```
const model = new JSONModel();
model.setSizeLimit(10_000);
```

Listing 7.9 Use of a Magic Number

The number `10_000` in the code shown in Listing 7.9 sets a maximum limit for list bindings on the created `JSONModel`. Because of the associated setter call, even though it's relatively clear what the number is doing, that value might also be used in other places of the code to also set the same limit in other models. Moving the number to a named constant not only makes it easier to refer to the same number but also makes changing the limit if needed easier, as shown in Listing 7.10.

```
const MAX_ENTRIES = 10_000;
// ...

const model = new JSONModel();
model.setSizeLimit(MAX_ENTRIES);
```

Listing 7.10 Use of a Named Constant

For numbers, 0 or 1 are usually the only literals that should appear directly in the code. Consider moving any other value to a named constant.

In the following sections, we'll look at different literal types for numbers, strings, Booleans, regular expressions, arrays, and objects. We'll also explore the values null and undefined.

7.2.1 Numbers

Numbers in JavaScript can be of type `number` or `bignum`. The type `number` is a double-precision floating point type with 64 bits and includes integers and floating point numbers. The `bignum` type is used for integers that are arbitrarily big.

Many ways exist for writing number literals, and you should choose the way that makes the number easier to read and understand for your use case. We'll describe recommendations for integer literals and floating point numbers, as well as consider numeric separators and exponentials, in this section.

Integer Literals and Floating Point Numbers

Integer literals can be written in different bases. The most common base is base 10, or decimal, but syntaxes exist for octals (base 8), hexadecimals (base 16), and even binaries (base 2). Floating point literals can only be written directly with the decimal base syntax, using a dot (.) as a decimal separator. Let's look at these various syntaxes with the examples shown in Listing 7.11.

```
// decimals
0
1
1000
-1234
0999

// floating point numbers
0.1
123.0001
-.0002 // same as -0.0002
.2134 // same as 0.2134
```

```
// octals
0555 // 365 in decimal, non-strict mode
0o555 // 365
00123 // 83
0o1 // 1
-0o765 // -501

// hexadecimals
0xCAFEBABE // 3405691582
0xfffffff // 16777215
-0X1234 // -4660

// binary
0b1111 // 15
0b0000000 // 0
0b10 // 2
-0B1010 // -10
```

Listing 7.11 Different Number Literals

Floating point numbers can omit the digit before the dot if that digit is zero; however, a good practice is to always write the digit zero, which also makes the literal easier to read (i.e., `-0.1` as opposed to `-.1`).

Apart from decimals, special prefixes are used for the different bases: `0o` for octals, `0x` for hexadecimals, and `0b` for binaries. The letter on the prefix can be written either in lowercase like `0x` or uppercase like `0X`. However, you should prefer using the lowercase version since it presents an easier-to-spot separation between the prefix and the number. Octals only accept digits from 0 to 7; binaries, only 0 and 1; and hexadecimals, only digits from 0 to 9 and letters from A to F, either in uppercase or lowercase.

Decimals can be written simply like `1000` or `-42`. In *non-strict mode*, octals can be written with a leading zero, like `0555` for the decimal `365`. However, if the number after the zero contains digits greater than or equal to 8, that number will be interpreted as a decimal, like `0118`. The `flight` object shown in Listing 7.12 prefixes a zero to the `flight` number, `043`, which is actually an octal literal parsed as the number `35`, a completely different flight!

```
const flight = {
  carrier: 'AC',
  flightNumber: 043, // parsed as the number 35
  from: 'DUB',
  to: 'YYZ',
};
```

Listing 7.12 Flight Number as an Octal Literal

Using octals in code is rather rare. *Strict mode* prohibits this octal syntax with a leading 0 and throws a `SyntaxError` when encountered. If you stick to strict mode for all your modules, you won't have to worry about this issue, as shown in Listing 7.13.

```
sap.ui.define([], function () {
  'use strict';

  // ...

  const flight = {
    carrier: 'AC',
    flightNumber: 043, // SyntaxError!
    from: 'DUB',
    to: 'YYZ',
  };

  // ...
});

});
```

Listing 7.13 Octal Literal with a Leading Zero Prevented in Strict Mode

Always use strict mode in your modules with the `'use strict'`; line at the start of the module function. This mode prevents many types of surprising issues in your code, like the octal syntax shown earlier in Listing 7.12.

Integers that are too big usually use the exponential notation, which we'll describe later in this section. However, if precision is important, you can use big integers to represent arbitrarily large integers. A big integer literal uses the lowercase letter `n` as a suffix, as in the literal `1_234_567_890_123_456_789_012_345_678_901n`. This prefix creates a number of type `bigint`, as opposed to the usual `number` for all other numbers. Big integers cannot participate in operations with other numbers, and you'll have to convert all numbers to big integers to work with them.

Big integer literals can be used with any number base literal, like a hexadecimal or a binary. For example, globally unique identifiers (GUIDs) are defined as 128-bit integers and thus can't be stored in the 64 bits of the `number` type. Instead, a big integer can be used to store a GUID precisely, like in the literal `0xda76f60e_8453_4eaa_bd60_b65701f63894n`.

Also be aware of the global variables `Nan` (and the `isNaN` function), for not-a-number, and `Infinity` (and `-Infinity`), which can be used or returned from number processing functions or expressions. A prominent example is a divide-by-zero scenario, which results in `Infinity` instead of an error: `1 / 0 === Infinity` is true. Always test for limits and boundary conditions in code that engages in number crunching or parsing.

Numeric Separators

Both big and not-so-big numbers can be hard to read, and you might lose sense of their magnitude or grouping of digits. As shown in earlier examples, you can use an underscore `_` as a numeric separator. This separator can be used in any position *within* the number.

For decimals, using the underscore as a thousands separator make sense, as in `10_000, 9_007_199_254_740_991` (which is the `Number.MAX_SAFE_INTEGER`) or `1_234_567.123456`. An underscore can be confusing in other positions, like `1_0000` or `1_2_3_4`, or even after the decimal separator like in `1_234.123_456_789`. Big integers can also use underscores, as in `1_234_567_890_123_456_789_012_345_678_901n`.

For hexadecimal or binary numbers, you can use underscores to separate different groups of digits. For example, you can group a hexadecimal number by byte like in `0x25_98_CA_7A_70` (each digit is 4 bits) or a binary number by nibble (4 bits), like in `0b1101_0010_1011_1001`. Alternatively, you can group the digits of a big integer GUID, like `0xda76f60e_8453_4eaa_bd60_b65701f63894n`. This approach makes visually parsing the different groups within the number easier.

Exponentials

In Chapter 6, Section 6.10.5, we defined two physics constants to hold the mass of the Earth and the mass of the Sun, which was simplified in the code shown in Listing 7.14.

```
const Mass = {
  SUN_MASS_IN_KG: 1.989e+30,
  EARTH_MASS_IN_KG: 5.972e+24
};
```

Listing 7.14 Numbers Using Exponential Syntax

This kind of literal using powers of 10 is commonplace when writing very large or very small numbers. This kind of literal is also a way to use scientific notation in JavaScript.

An *exponential number* uses a syntax that starts with a base number, which is either an integer or a floating point number, followed by the `e` character, followed by the *exponent*, which is a signed integer that denotes the number of powers of 10 that the base number should be multiplied by. So, in our example, the literal `1.989e+30` can be read as its scientific notation counterpart 1.989×10^{30} , which is the number `1,989,000,000,000,000,000,000,000,000` (at almost two nonillion kilograms, the Sun is mind-bogglingly heavy). The sign for a positive exponent is optional, and `1.989e+30` can also be written as `1.989e30`. The `e` character can also be written in uppercase, as in `1.989E30`, but we prefer lowercase, which is easier to spot.

Use the exponential number syntax for numbers that are either too big or too small and when scientific notation makes sense and absolute precision is not required. To make your code adhere to scientific notation, another important rule is to always use a

single non-zero digit to the left of the decimal point (if there is one). Some examples of exponential number literals are shown in Listing 7.15.

```
// adhering to the scientific notation
6.67384e-11 // gravitation constant
2.99792458e8 // speed of light
1e4 // 10000
1e-4 // 0.0001
-1e-4 // -0.0001
1.234e3 // 1234

// NOT adhering to the scientific notation, don't use!
10e3 // 10000, should be written as 1e4
1989e27 // should be 1.989e30
0.1989e31 // should be 1.989e30
0.1e3 // 100, should be 1e2
0e10 // 0
0e-30 // 0
```

Listing 7.15 Exponential Number Literals

Additionally, for numbers that can be comfortably written without exponents, consider writing the number in full, optionally using numeric separators, as shown in Listing 7.16.

```
300_000 // instead of 3e5
10_000 // instead of 1e4
0.001 // instead of 1e-3
```

Listing 7.16 When Not to Use Exponents

7.2.2 Strings

String literals can be written in three ways: using single quotes, double quotes, or backticks as delimiters. Single quotes and double quotes are older and are equivalent. Backticks are newer and support strings spanning multiple lines in the source code and also support *embedded expressions* delimited by \${ and } that are resolved at runtime, a feature also sometimes called *string interpolation*. Strings with backticks are called *template literals*. Let's look at the examples shown in Listing 7.17.

```
// using single-quotes
const name = 'Charles Babbage';

// using double-quotes
const invention = "Mechanical Computer";
```

```
// multiline string must use backticks
const quote = `On two occasions I have been
asked, "Pray, Mr. Babbage, if you put into the
machine wrong figures, will the right answers
come out?" I am not able rightly to apprehend
the kind of confusion of ideas that could provoke
such a question.`;
```

```
// embedded expressions must use backticks
const formattedQuote = `${quote} - ${name}`;
```

Listing 7.17 String Literals

Strings that don't span multiple lines and that don't use embedded expressions should be written consistently. You might prefer single quotes or double quotes or even backticks. This book mostly uses single quotes. In your own project, enforce a consistent quoting style but allow for deviations that improve readability. ESLint has a rule that can check for consistency in quote usage: quotes, which defaults to double quotes. The rule also has the option avoidEscape that, when true, allows a different quote mark to be used if the string contains a quote that would have to be escaped otherwise. Check out the examples shown in Listing 7.18.

```
/*eslint quotes: ['error', 'single', { avoidEscape: true }]*/
// using single-quotes by default
const name = 'Guybrush Threepwood';

// double-quotes allowed because the string contains a single-quote
const intro = "I'm a mighty pirate!";

// escaping the quote makes the string harder to read
const pitch = 'I\'m selling these fine leather jackets.';
```

Listing 7.18 Enforcing Single Quotes While Allowing Double Quotes When Needed

You should use backticks with embedded expressions instead of string concatenation to create dynamic strings, as shown in Listing 7.19.

```
function readProduct(productId) {
  return new Promise((resolve, reject) => {
    const model = this.getView().getModel();
    model.read(`/ProductSet('${productId}')`, {
      success: resolve,
      error: reject
    });
  });
}
```

```
});  
});  
}
```

Listing 7.19 Using an Embedded Expression for an OData Endpoint

As shown in Listing 7.19, the OData endpoint to read a single product can be created using a string with an embedded expression. Creating the same string using string concatenation would be harder to read, like in `"/ProductSet('' + productId + '')"`. Multiple strings must be defined, and getting lost with all those opening and closing quotes can be confusing. This confusion is exacerbated when the final string is composed of multiple components, for example, if your OData entity has a key composed of multiple properties, as shown in Listing 7.20.

```
// concatenation is harder to write and to read  
"/OrderItem(OrderId='' + orderId + '',ItemNumber='' + itemNumber + '')"  
  
// prefer interpolation  
'/OrderItem(OrderId='${orderId}',ItemNumber='${itemNumber}')'
```

Listing 7.20 String Concatenation versus String Interpolation

Template literals also allow for *tagged templates*, where the interpretation of the template literal is performed by the *tag function* being used. This is a special syntax that adds the function name to the beginning of the template literal. Check out the example shown in Listing 7.21.

```
function process(transform, strings, ...values) {  
  return strings.reduce((prev, next, i) =>  
    `${prev}${next}${transform(values[i] ?? '')}` , '' );  
}  
  
function scream(...args) {  
  return process(value => String(value).toUpperCase(), ...args);  
}  
  
const howMany = 5e3;  
const what = 'exo-worlds';  
const message = scream`Hello ${howMany} ${what}!`;  
console.log(message); // Hello 5000 EXO-WORLDS!
```

Listing 7.21 Using a Tagged Template Literal

As shown in Listing 7.21, the tag function `scream` is used in the tag template literal that follows it. This function converts any embedded expressions in the literal to uppercase. The syntax, where the function name is directly followed by a template literal with no

parentheses, is quite different from usual function calls and might seem surprising to many developers. Use it with care and only in cases where a tag template makes sense, like when creating domain-specific languages embedded in JavaScript.

The tag function itself is a normal function that follows a specific form: It takes as parameters (1) an array with the different string parts of the template literal and (2) a stream of parameters where each element is the result of each embedded expression in the literal, which can be declared as a rest parameter. The usual definition could be something like `function tag(strings, ...values)`. The `scream` function is using a more generic `process` function that applies a `transform` function to each embedded expression value. Note that embedded expressions can be of any type, not just strings, and so the `transform` function used in `scream` first calls the `String` function to then be able to call `toUpperCase` on the resulting string. Finally, a tag function can return a value of any type, not just a string.

Like magic numbers, check that your string literals are not being used as “magic strings” and refactor them to use named constants or enums if necessary. When strings are meant to be displayed to the end user, like with labels, messages, and titles, they should be added to the `i18n.properties` file (`i18n` is an abbreviation for *internationalization*). The code then refers to an `i18n` string by an alias (its key), and it can be reused and translated into multiple languages. It even supports its own string substitution syntax, with positional placeholders like `{0}`, `{1}`, `{2}`, and so on.

In Chapter 2, Section 2.1.7, we showed many examples of reading strings from the `i18n.properties` file. They can be used in the code and also in XML view files, usually accessed through the special “`i18n`” named model (an `sap.ui.model.resource.ResourceModel` instance), that can be set up in the `manifest.js` file.

String literals (but not template literals) can also be used as object keys and when accessing object properties or functions. Take a look at the code shown in Listing 7.22 for some examples.

```
const employee = {  
  'name': 'Guybrush Threepwood',  
  'profession': 'Pirate?',  
  'favorite-hobby': 'Sword Fighting',  
  'utter-greeting'() {  
    return "Hi! My name's Guybrush Threepwood, and I want to be a pirate!";  
  },  
};  
  
const name = employee['name'];  
const hobby = employee['favorite-hobby'];  
const greeting = employee['utter-greeting']();
```

Listing 7.22 Strings in Object Property Definition and Access

However, following the usual naming conventions (refer to Chapter 6), object keys should be in camelCase or PascalCase, which don't require quotes around them. So, the need to use string literals as object keys or properties should be rare, and you should only use string literals if you have keys that cannot be expressed as JavaScript identifiers. Perhaps, you need an object with special keys to interoperate with external services or application programming interfaces (APIs) that require them.

ESLint has two rules you can use to check for these situations: quote-props and dot-notation. The example code shown in Listing 7.22 should then be rewritten, as shown in Listing 7.23, where we also show the preferred configuration for those ESLint rules.

```
/*eslint dot-notation: 'error'*/
/*eslint quote-props: ['error', 'as-needed']*/

const employee = {
  name: 'Guybrush Threepwood',
  profession: 'Pirate? ',
  'favorite-hobby': 'Sword Fighting',
  'utter-greeting' () {
    return "Hi! My name's Guybrush Threepwood, and I want to be a pirate!";
  },
};

const name = employee.name;
const hobby = employee['favorite-hobby'];
const greeting = employee['utter-greeting']();
```

Listing 7.23 Using Strings in Object Properties Only When Needed

7.2.3 Booleans

Boolean literals are simply true and false. However, you must know what values evaluate to either true or false, so that Boolean tests in the code can be written more concisely. We explored Booleans at large in Chapter 2, Section 2.1.11. Let's look at the code shown in Listing 7.24 for some examples.

```
function revealTheTruth(value) {
  console.log (!!value);
}

// all "falsy" values
revealTheTruth(false);
revealTheTruth(0);
revealTheTruth('');
revealTheTruth(null);
```

```
revealTheTruth(undefined);
revealTheTruth(NaN);

// everything else is "truthy"
revealTheTruth(true);
revealTheTruth(1e-308);
revealTheTruth(' '); // a space
revealTheTruth({});
revealTheTruth([]);
revealTheTruth(Infinity);
```

Listing 7.24 The Truth Shall Set You Free

To convert an object or a primitive to a Boolean, the code shown in Listing 7.24 uses double negation, like in `!!value`. The Boolean function could also have been used, as in `Boolean(value)`.

You should rarely need to compare values directly with either `true` or `false` or with the list of *falsy* values when checking for existence or non-emptiness, as shown in Listing 7.25. Only check for explicit values like `false` or `undefined` if you really need the variable to be that specific value, and not generally for it to be *truthy* or *falsy*.

```
if (isActive) {}
// instead of:
// if (isActive === true) {}

if (!person) {}
// instead of:
// if (person === null) {}
```

```
function printCoord(coord) {
  if (!coord) {}
  // instead of:
  // if (coord === undefined) {}
}
```

Listing 7.25 Testing Boolean Variables

7.2.4 Regular Expressions

Regular expressions were designed to be extremely concise and powerful at expressing complex patterns of text. Because of this goal, regular expressions are notoriously hard to write and read. Modern integrated development environments (IDEs) and code editors do a good job of highlighting the different parts, groups, and special characters of regular expression literals, but a good understanding of regular expression syntax is

still necessary. A number of great tutorials are available online like Regex Learn (<https://regexlearn.com>) as are regex testers like regex101 (<https://regex101.com>) and visualizers like Regex Tester (<https://extendsclass.com/regex-tester.html>), which do a great job of visually breaking down and explaining all parts of a regex.

You can use a regular expression literal or the `RegExp` constructor to create regular expressions, as shown in Listing 7.26.

```
function rx(...args) {
  // start or end slashes with optional flags
  const boundary = /(^\//|(\/[a-z]*$)/g;
  return process(
    regex => String(regex).replace(boundary, ''),
    ...args);
}

// yyyy-mm-dd
const datePattern = /\d{4}-\d{2}-\d{2}/;
// hh:mm:ss
const timePattern = /\d{2}:\d{2}:\d{2}/;

// a line in a log file: timestamp followed by an event
const linePattern = new RegExp(
  rx`(${datePattern}T${timePattern}) - (.* )`, 'gm');
```

Listing 7.26 Regex Literals and the `RegExp` Constructor

As shown in Listing 7.26, three regular expression literals exist: `boundary`, `datePattern`, and `timePattern`. The `linePattern` literal is a dynamic regular expression created with the `RegExp` constructor using the `rx` tagged template literal. `rx` helps with embedding regular expressions inside a template string and uses the `process` function, as shown previously in Listing 7.21.

Comments help clarify what each regular expression matches, either through an explanation text or by laying out the expected pattern. Being rather dense and usually hard to read, regular expressions can benefit from well-placed comments. Refer to Chapter 11 for more information on comments.

The `RegExp` constructor takes a string and should only be used when your regex has dynamic parts or when you're creating a regex based on a string that you did not create. String literals in the `RegExp` constructor can be awkward because backslashes must be escaped (but this requirement can be alleviated with the `String.raw` tagged template literal) and will not be highlighted like a regular expression literal by code editors. Use the ESLint rule `prefer-regex-literals` to check that you're using literals when possible.

7.2.5 Arrays

Array literals are enclosed in square brackets [and], as shown in Listing 7.27.

```
const numbers = [1, 2, 4, 5];

const employees = [
  { name: 'Roger Wilco' },
  { name: 'Guybrush Threepwood' },
];
```

Listing 7.27 Array Literals

Arrays should usually be employed to store a collection of items of the same type, like strings or objects with the same properties. You can more easily process the elements of an array in a generic way. An array can have a dangling comma at the end, like in the last item in the `employees` array shown in Listing 7.27. Consider leaving a dangling comma if the array spans multiple lines like with `employees`. With this dangling comma, you can more easily add items or change the order of items in the array without having to muck about with trailing commas.

If your array does not have duplicate items and the order does not matter, consider using a `Set` instead. A `Set` has methods to add or delete entries that are far more convenient than using array methods like `splice`.

Array destructuring syntax can be used to unpack an array's elements into separate variables or parameters, like in Listing 7.28.

```
const logFile =
  2022-04-04T09:30:31 - Door opened
  2022-04-04T09:34:21 - Door closed`;

let match;
while (match = linePattern.exec(logFile)) {
  const [, when, event] = match;
  console.log(`"${event}" at ${when.replace('T', ' ')}`);
}
```

Listing 7.28 Array Destructuring

As shown in Listing 7.28, the regular expression `linePattern` (defined earlier in the code shown in Listing 7.26) matches the lines of `logFile`. The `while` loop captures each `match` and uses array destructuring to unpack the regular expression groups that were matched. The `match` array first element is the full line that was matched, and the subsequent elements are the groups defined in the regular expression enclosed in the parentheses (and). We're not interested in the first element, only the second and the third,

and so the syntax `[, when, event]` skips the first element (the first comma); captures the second as the `when` variable; and captures the third as the `event` variable. This syntax is flexible and accepts both default values and the spread syntax to capture all remaining elements in an array. Let's look at some more examples, shown in Listing 7.29.

```
const numbers = [1, 2, 3];

// skipping an element
const [one,, three] = numbers;

// capturing the rest using the spread syntax
const [first, ...rest] = numbers;
console.log(rest); // [ 2, 3 ]

// using a default
const [uno, due, tre, quattro = 4] = numbers;
console.log(uno, due, tre, quattro); // 1, 2, 3, 4

// a function with destructuring and defaults
function printPoint([x = 0, y = 0] = []) {
  console.log(x, y);
}

printPoint();          // 0 0
printPoint([]);        // 0 0
printPoint([42]);      // 42 0
printPoint([42, 44]);   // 42 44
```

Listing 7.29 Arrays: Destructuring, Defaults Values, and the Spread Syntax

The spread syntax (`...,` discussed in detail in Chapter 2, Section 2.1.7), can also be used as a shorthand to create a shallow copy of an array or to concatenate two or more arrays, without changing the original array, as shown in Listing 7.30. This syntax is easier to read and should be preferred over the array methods `slice` for copying and `concat` for concatenating arrays. This syntax also works with any iterable, not just with arrays.

```
const odds = [1, 3, 5];

// preferred over odds.slice()
const copy = [...odds];

const evens = [2, 4, 6];
```

```
// preferred over odds.concat(evens)
const combined = [...odds, ...evens];
```

Listing 7.30 Copying and Concatenating Arrays with Spread Syntax

7.2.6 Objects

Object literals are key-value pairs enclosed in curly braces `{ and }`, as shown in Listing 7.31.

```
const point = { x: 12, y: -23 };

const pirate = {
  firstName: 'Guybrush',
  lastName: 'Threepwood',
}
```

Listing 7.31 Object Literals

As with arrays, for object literals spanning multiple lines, like `pirate` shown in Listing 7.31, consider leaving a dangling comma after the last key-value pair to make changes easier. The keys in object literals and objects should always be of type `string` or `symbol`, but the values can be of any type. Keys can also be dynamically created (computed properties enclosed in brackets `[and]`), as shown in Listing 7.32. If you need a real map or dictionary that supports keys of arbitrary types, use the `Map` type instead.

```
const suffix = 'Name';

const names = [
  `first${suffix}`: 'Guybrush',
  `last${suffix}`: 'Threepwood',
  *[Symbol.iterator]() {
    yield this[`first${suffix}`];
    yield this[`last${suffix}`];
  },
};

for (const name of names) {
  console.log(name);
}
```

Listing 7.32 Computed Properties

Object literals also support shorthand definitions, where a variable defines the key name and the value, as shown in Listing 7.33. When creating objects based on variables

with the same name as the object's keys, use shorthand properties to make the code more concise and easier to read.

```
function makePerson(firstName, lastName) {
  return { firstName, lastName };
}

const person = makePerson('Guybrush', 'Threepwood');
```

Listing 7.33 Shorthand Properties

As shown in Listing 7.33, shorthand syntax is used in the object literal `{ firstName, lastName }`. Without this shorthand syntax, the code would be both redundant and harder to read: `{ firstName: firstName, lastName: lastName }`.

As with arrays, destructuring can also be used to unpack an object's values by its keys. Destructuring is particularly useful for only assigning the needed properties of an imported object or an object parameter. Look at the examples shown in Listing 7.34. Refer to Chapter 5, Section 5.8.4, for more on destructuring function parameters.

```
const person = {
  firstName: 'Guybrush',
  lastName: 'Threepwood',
  fullName: 'Guybrush Threepwood',
  profession: 'Pirate? ',
  // .... lots of other properties
}

// only interested in the first and last names
const { firstName, lastName } = person;

// a function with destructuring and defaults
function printName({ fullName = 'John Doe' } = {}) {
  console.log(fullName);
}

printName(); // John Doe
printName({}); // John Doe
printName({ x: 1, y: 2 }); // John Doe
printName(person); // Guybrush Threepwood
```

Listing 7.34 Object Destructuring

The spread syntax can also be used with objects to capture all keys that are not already captured in the literal. This feature is useful for creating a shallow copy of an object or

for combining two or more objects into a single object, as shown in Listing 7.35, where `copy` is a shallow copy of `person` and `combined` combines the `person` and `position` objects.

```
const person = {
  firstName: 'Guybrush',
  lastName: 'Threepwood',
};

// shallow copy
const copy = { ...person };

const position = {
  profession: 'Pirate?',
  seniority: 'Freshman',
};

// combined object
const combined = { ...person, ...position };
```

Listing 7.35 Object Spread Syntax

7.2.7 Null and Undefined

The literals `null` and `undefined` are special values that are *falsy* and generally mean that an object is missing or not defined. Let's look at the examples shown in Listing 7.36.

```
let name;
console.log(name); // undefined
console.log(typeof name); // undefined

name = null;
console.log(name); // null
console.log(typeof name); // object

function printValue(value) {
  console.log(value);
}
printValue(); // undefined

const person = { firstName: 'Guybrush' };
printValue(person.firstName); // Guybrush
printValue(person.lastName); // undefined
```

Listing 7.36 Null and Undefined

The value `undefined` has its own type, also named `undefined`. The value `null` is of type `object`. `undefined` is special and indicates a variable, a parameter, or an object property has not yet been defined. A variable or parameter may have been declared but not initialized (like name before the assignment to `null`) or provided as a parameter (like the value parameter). An object property may simply not exist in the object (like `person.lastName`).

Never Assign `undefined` to a Variable, Object Property, or Parameter

Initializing a variable, initializing an object property, or passing a parameter explicitly as `undefined` are all conceptually *lies*. You're actually defining the value to be `undefined`! So, don't do it. When you need to define a value to be empty or missing, use `null` instead. You can also leave a variable uninitialized or an object property undeclared, which will result in it being `undefined`. Consider the examples shown in Listing 7.37.

```
// variable:  
let name;  
// or:  
let name = null;  
// but not:  
let name = undefined;  
  
// object property:  
const person = {  
  firstName: 'Guybrush',  
};  
// or:  
const person = {  
  firstName: 'Guybrush',  
  lastName: null,  
};  
// but not:  
const person = {  
  firstName: 'Guybrush',  
  lastName: undefined,  
};  
  
// parameter passing:  
reticulateSplines();  
// or:  
reticulateSplines(null);  
// with null before a non-null parameter  
reticulateSplines(null, 42);  
// but not:  
reticulateSplines(undefined);
```

```
// and not:  
reticulateSplines(undefined, 42);  
  
// undefining an object property  
delete person.firstName;  
// but not:  
person.firstName = undefined;
```

Listing 7.37 Avoiding `Undefined` Assignments

7.3 Summary

In this chapter, you learned how to declare and use variables, and we covered different types of literals and the necessary syntax to create and use literals.

We started with variables and scoping rules and explored the differences between variables declared with `var`, `let`, and `const`. The keyword `var` shouldn't be used anymore, and `const` should be preferred over `let`.

We then moved on to talk about literals. Starting with number literals, we covered different syntaxes for various number bases, big integers, numeric separators, and exponentials. Moving on to strings, which can be defined with different quote marks, as template literal strings and as tagged template literal strings using a tag function. We then talked about Booleans and how different values can be interpreted as Booleans. Regular expressions came next, with its funky but powerful syntax. We then described arrays and object literals, including destructuring assignments and the spread syntax. Finally, we looked into the literals `null` and `undefined` and highlighted how `undefined` should not be used as an assignment value.

In the next chapter, we'll talk about how to use control flow statements in your projects.

Contents

Preface	17
---------------	----

1 Introduction 23

1.1 What Is Clean SAPUI5?	23
1.1.1 What Is Readability?	24
1.1.2 What's the Story behind Clean SAPUI5?	25
1.2 How to Get Started with Clean SAPUI5	26
1.3 How to Handle Legacy Code	27
1.4 How to Check Code Automatically	29
1.5 How Does Clean SAPUI5 Relate to Other Guides?	29
1.6 Summary	30

2 JavaScript and SAPUI5 31

2.1 JavaScript ES6+ Features	32
2.1.1 Browser Support	32
2.1.2 const Keyword	34
2.1.3 Hoisting and Function Scoping with var	38
2.1.4 Block Scoping with let and const	39
2.1.5 Arrow Functions	39
2.1.6 Template Literals	42
2.1.7 Spread Syntax	44
2.1.8 Destructuring Assignment	57
2.1.9 Promises, async, and await	62
2.1.10 for await of Statements	85
2.1.11 Default Parameters	88
2.1.12 Classes	90
2.1.13 Modules	102
2.2 TypeScript	106
2.2.1 Transpiling and Source Mapping	108
2.2.2 Strong Typing	109
2.2.3 Features	110
2.3 Summary	117

3 Project Structure	119
3.1 Components in SAPUI5	119
3.2 Important Artifacts	122
3.2.1 Component Controller	122
3.2.2 Descriptor	128
3.2.3 Root View	129
3.2.4 The index.html File	129
3.3 Freestyle Applications	129
3.4 SAP Fiori Elements	132
3.4.1 Floorplans	134
3.4.2 Application Generation	137
3.5 Library Projects	139
3.5.1 The library.js File	139
3.5.2 SAPUI5 Reuse Components	140
3.5.3 Folder Structure	140
3.6 Model-View-Controller Assets	142
3.7 Summary	144
4 Modules and Classes	147
4.1 Controller Inflation	148
4.1.1 Model-View-Controller Pattern	148
4.1.2 Object View	152
4.1.3 Dialogs	174
4.2 Module Lifecycle	179
4.2.1 Execution Context	180
4.2.2 Loading and Caching	181
4.2.3 Single-Page Application Lifecycle	184
4.3 Reusability and Testability	187
4.3.1 Defining an Interface	188
4.3.2 Module State	189
4.3.3 Documenting	192
4.3.4 Unit Testing	193
4.3.5 Dependency Mocking	193

4.4 Service Modules versus Class Modules	197
4.4.1 Service Modules	198
4.4.2 Class Modules	200
4.5 Summary	204
5 Functions	205
5.1 Function Definition	205
5.2 The Function Object	206
5.3 Instance Methods	208
5.4 Event Handlers and Callbacks	211
5.5 Callback Execution Context	211
5.6 Getters and Setters	213
5.7 Anonymous Functions	215
5.8 Function Parameters	217
5.8.1 Function Arity	217
5.8.2 Boolean Parameters	222
5.8.3 Rest Parameters	225
5.8.4 Destructuring Parameters	226
5.8.5 Default Values	228
5.9 Promises	230
5.10 Generators	236
5.11 Function Body	237
5.11.1 Do One Thing	238
5.11.2 Descend One Level of Abstraction	239
5.11.3 Keep Functions Small	242
5.12 Invoking Functions	245
5.13 Closures	247
5.14 Summary	248
6 Naming	249
6.1 Descriptive Names	249
6.2 Domain Terms	251

6.3	Design Patterns	252
6.4	Consistency	253
6.5	Shortening Names	254
6.6	Noise Words	255
6.7	Casing	255
6.8	Classes and Enums	257
6.9	Functions and Methods	258
6.9.1	SAPUI5 Common Methods	259
6.9.2	Event Handlers	259
6.10	Variables and Parameters	260
6.10.1	Booleans	260
6.10.2	Loop Variables	261
6.10.3	Comparison Functions and Parameters	264
6.10.4	Event Parameters	264
6.10.5	Constants	264
6.11	Private Members	265
6.12	Namespaces	267
6.13	Control IDs	268
6.14	Hungarian Notation	270
6.15	Alternative Rules	271
6.16	Summary	273

7	Variables and Literals	275
7.1	Variables	275
7.2	Literals	279
7.2.1	Numbers	280
7.2.2	Strings	284
7.2.3	Booleans	288
7.2.4	Regular Expressions	289
7.2.5	Arrays	291
7.2.6	Objects	293
7.2.7	Null and Undefined	295
7.3	Summary	297

8 Control Flow

8.1	Conditionals	299
8.1.1	if, else if, and else	300
8.1.2	switch	302
8.2	Loops	303
8.2.1	while Loops	303
8.2.2	do...while Loops	303
8.2.3	for Loops	304
8.2.4	for...of Loops	305
8.2.5	for...in Loops	305
8.3	Conditional Complexity	306
8.3.1	Avoid Empty if Branches	306
8.3.2	Construct Positive Conditions	307
8.3.3	Decompose Complex Conditions	307
8.3.4	Encapsulate Conditionals	308
8.3.5	Avoid Flags as Parameters	309
8.3.6	Avoid Nested if Statements	310
8.3.7	Prefer Declarative Style over Imperative Style	312
8.4	Summary	313

9 Error Handling

9.1	throw and try/catch Statements	315
9.2	Using Error Objects	317
9.3	Error Handling Using Messages	318
9.4	Error Handling Using Controls	320
9.5	Error Handling Best Practices	323
9.6	Summary	327

10 Formatting

10.1	Motivation	329
10.2	Vertical versus Horizontal Formatting	330
10.2.1	Vertical Formatting	330
10.2.2	Horizontal Formatting	336

10.3 Indentation	339
10.4 XML Views	341
10.4.1 Vertical Distance and Properties	342
10.4.2 Aggregations	344
10.4.3 Bindings	345
10.5 Additional Considerations	347
10.5.1 Tabs versus Spaces	347
10.5.2 Line Endings	348
10.5.3 Dangling Commas	348
10.5.4 Ternary Operators and Inline Logic	349
10.6 Formatting for TypeScript in SAPUI5	351
10.6.1 Types versus Interfaces	351
10.6.2 Classes	353
10.6.3 Definition Files	355
10.6.4 Namespaces and Modules	355
10.6.5 Enums	358
10.6.6 Import Statements	360
10.6.7 Suggested Formatting Standards	360
10.7 Building and Maintaining a Code Style Guide	361
10.8 Formatting Tools	363
10.8.1 Formatting versus Linting	363
10.8.2 EditorConfig	365
10.8.3 Prettier	365
10.9 Summary	368
11 Comments	369
11.1 Express Yourself in Code	369
11.2 The Good: Comment Placement and Usage	371
11.2.1 Intention-Revealing Comments	371
11.2.2 Summary Comments	374
11.2.3 Comments on Variables and Parameter Definitions	376
11.2.4 Parameter Name or Argument Comments	377
11.2.5 JSDoc Comments	378
11.3 The Bad: Comments to Avoid or Refactor	381
11.3.1 Formatting and Alignment	382
11.3.2 Sections and Separators	385

11.3.3 Commented Out Code	387
11.3.4 No-Code Comments	389
11.3.5 Closing Brace Comments	389
11.3.6 Meta-Comments	390
11.4 The Ugly: Special Comments	391
11.4.1 Legal Comments	391
11.4.2 To-Do and Other Code Tags	392
11.4.3 Pragma Comments	393
11.5 Summary	394
12 Static Code Checks and Code Metrics	397
12.1 Linting	399
12.1.1 JavaScript Linters	400
12.1.2 XML Linting	414
12.2 Code Metrics	417
12.2.1 Cyclomatic Complexity	417
12.2.2 Nesting	422
12.2.3 Function Parameters	428
12.2.4 Function Length	430
12.2.5 Fan-In/Fan-Out	431
12.2.6 Code Repetition	433
12.2.7 Maintainability	435
12.3 Summary	437

13 Testing	439
13.1 Principles	440
13.1.1 Write Testable Code	440
13.1.2 Test Pyramid	443
13.1.3 FIRST Principles of Test Automates	444
13.1.4 Test-Driven Development	445
13.1.5 Enable Mocking?	446
13.1.6 UIVeri5 versus OPA5 versus QUnit	447
13.1.7 Page Objects	448
13.1.8 Don't Obsess about Coverage	450
13.1.9 Readability Rules	451

13.2	Code under Test	451
13.2.1	Meaningful Names for Code under Test	452
13.2.2	Extract the Call to Code under Test into Its Own Test Method	452
13.3	Injection	453
13.3.1	Constructor Injection	454
13.3.2	Setter Injection	454
13.3.3	Sinon.JS	455
13.3.4	Prototype	456
13.3.5	OData V2 Mock Server	457
13.4	Test Methods and Journeys	458
13.4.1	Test Method Names	458
13.4.2	Journeys: The Larger Context of Testing	459
13.4.3	Journey Steps: What Is Being Done	459
13.4.4	Using Given-When-Then	460
13.5	Test Data	461
13.5.1	Make It Easy to Spot Meaning	461
13.5.2	Make It Easy to Spot Differences	462
13.5.3	Use Constants to Describe the Purpose and Importance of Test Data	462
13.5.4	Test Data for OData V2 Mock Server	463
13.6	Assertions	465
13.6.1	Limit Yourself to a Few Focused Assertions	465
13.6.2	Use the Right Assert Type	466
13.6.3	Assert Content, Not Quantity	466
13.6.4	Assert Quality, Not Content	467
13.6.5	Check for Expected Exceptions	467
13.6.6	Write Custom Assertions to Shorten Code and Avoid Duplication	468
13.6.7	Display Meaningful Error Messages in Case of Timeouts for Page Objects	469
13.7	Summary	470
14	TypeScript and Related Technologies	471
14.1	TypeScript	471
14.1.1	Current State	472
14.1.2	Using SAPUI5 with TypeScript	473
14.1.3	Adding TypeScript to an SAPUI5 App	474
14.1.4	Usage Examples	475
14.1.5	How to Stay Updated	485

14.2	Web Components	485
14.2.1	Current State	486
14.2.2	Clean Code with Web Components	488
14.2.3	Usage Examples	489
14.2.4	How to Stay Updated	493
14.3	Fundamental Library	494
14.3.1	Fundamental Styles	494
14.3.2	Fundamental Library	494
14.3.3	Current State	498
14.3.4	How to Stay Updated	498
14.4	Summary	498
15	Implementing Clean SAPUI5	499
15.1	A Common Understanding among Team Members	499
15.2	Collective Code Ownership	500
15.3	Clean Code Developer Initiative	501
15.4	Tackling the Broken Window Effect	503
15.4.1	Static Code Analysis	505
15.4.2	Metrics	505
15.4.3	Code Coverage	506
15.5	Code Review and Learning	506
15.5.1	Code Review Prefixes	507
15.5.2	Style Guides	507
15.5.3	Making Practices Visible	507
15.5.4	Feedback Culture	507
15.6	Clean Code Advisors	509
15.7	Learning Techniques	510
15.7.1	Kata	510
15.7.2	Dojo	511
15.7.3	Code Retreats	511
15.7.4	Fellowship	512
15.7.5	Pair Programming	512
15.7.6	Mob Programming	513
15.7.7	Habits	513
15.8	Continuous Learning in Cross-Functional Teams	514
15.8.1	Profile of a Team Member	514
15.8.2	Cross-Functional Teams	515

Contents

15.8.3 Multipliers: Need for Topic Experts in the Team	516
15.8.4 Need for a Community of Practice	516
15.9 Summary	516
The Authors	517
Index	519

Index

- /annotations folder 133
/changes folder 134
/controller folder 121, 147
/extension folder 134
/i18n folder 131
/integration folder 147
/localService folder 131
/model folder 147
/root folder 130, 140
/src folder 141
/test folder 130, 147
/unit folder 147
/view folder 147
/webapp folder 130, 147
\$this model 168
-
- A**
- A.prototype object 91
Abstraction 114, 240, 433
 example 241
Abstract syntax tree (AST) 411
Aggregation 125, 166, 344
 bindings 345
Alignment 338
 break 384
 comments 383
Angular 496
Annotation 137
Anti-pattern 306, 429, 468
App.view.xml 129
Application lifecycle 184
Application programming interface (API) ... 46, 50, 62, 69, 104, 316
 browser 409
 typing 355
arguments object 40, 46, 53, 60, 377, 429
Array 42, 44, 52, 60, 291
 destructuring 291
 length 263
 nested 262
 promises 72
 questions and answers 86
 size 99
 spread syntax 45, 245
Array.concat 44
Array.prototype.flat method 46
- Arrow function 39, 206–207, 211, 216
 anonymous 217
 no parameter 218
 restrictions 40
 single parameter 218
 versus callback 217
- Artifact grouping 121
- Assembler 24
- Assertion 325, 465
 content, not quantity 466
 custom 468
 expected exceptions 467
 focus 465
 quality, not content 467
 type 466
- Assignment 337
- Association 332, 337
- async/await syntax 476
- Asynchronous code 62, 69, 441
 challenges 63
 linters 84
 loop 88
- Asynchronous JavaScript and XML (AJAX)
 callback 41
- Asynchronous module definition
 (AMD) 103, 473
- async keyword 62, 78–79, 82–83, 87, 107
- Atomic design 488
 principles 488
- Autocompletion 110
- Automated testing 439, 513
- Automatic test 387
- Automating style 330
- Automation 29, 398
- await keyword 62, 78–80, 82
-
- B**
- Babel 94, 105, 108, 351, 402, 473
- Backtick 284
- Base class 92, 94
- BaseController class 54
- BigInt 35, 280
- Binding 100, 345
 default mode 151
 expression 151, 158
 formatting 345

Binding (Cont.)
implement behaviors 154
one-way 150, 152
property 149
simple 150
test 160
two-way 152
Black grade 502
Block 39
nesting 424
scope 277
Boolean 35, 89, 222, 257, 259, 288
avoid 49, 223
enum 224
helper method 223
naming 260
Bootstrap 104
Bottlenecks 500
Boy Scout Rule 27
Branch coverage 506
Break statement 303
Broken window effect 401, 503
examples 504
Browser API 46
Browser support 32
Busy state 65, 82

C

Callback 41, 62, 65, 69, 87, 230, 245, 441
execution context 211
nesting 422
Callback hell 62, 66, 231, 422
asynchronous OData reads 64
camelCase 255, 258, 272, 288, 491
Can-I-use 33
case statement 420
Casing 255
catch handler 81
catch object 71–72
catch statement 316
Chaining 66, 72, 78
children property 94
Class 90, 97, 147, 266
base 92, 94
decorators 116
define 92
define purpose 188
error 318
extend 94
framework 97
inheritance 96

Class (Cont.)
inside TypeScript module 483
interfaces 114
keyword 90, 92–93, 96
mock 195
naming 257
private members 97
static methods 97
timer 213
TreeVisitor 196
TypeScript 353

Clean code advisor 509
Clean Code Developer Initiative 27, 501
Clean island 28
Clean SAPUI5 23, 25, 119
get started 26
legacy project 27
other guides 29

Closure 247–248
Code check 29, 397, 505
Code coverage 450, 505–506
Code metrics 397, 417
Code quality 399
Code repetition 433
Code retreat 510–511
Code review 362, 398, 506
feedback culture 507
managers 509
prefix 507
rules 508
visibility 507

Code rewrite 504
Code smell 468
Code steward 501
Code style guide 361
Code tag 392
Code under test 451
isolate call 452
meaningful names 452

Collective code ownership 500
steps 500

Comma 352
Comment 192, 346, 369, 372
alignment 382
argument 377
bad 381
break 384
closing brace 389
comment out code 387
good 371
intention-revealing 371
JSDoc 378

legal 391
no-code 389
placement 372
redundant 370
section 385
special 391
summary 374
to-do 387, 392
variables and parameters 376

CommonJS 105
Common understanding 499–500
Communication style 508
Community of practice 516
Comparison function 264
Component 252
test 443
Component.js 120, 122, 128, 181
Component controller 120, 122, 128
interface 123–124
Composing 168
Composite control development 164
Conceptual affinity 335
Condition 306, 349
break lines 350
encapsulate 308
guarding 310
negative and positive 307
Conditional check 311
Conditional complexity 306, 349
decompose 307
Conditional statement 299
Configuration file 407, 412
max cyclomatic complexity 422
Confirmation box 220, 235
connect method 114
Consistency 253, 256, 285, 330, 340, 347, 365
Constant 34, 264, 279
assignment 35
versus other variables 36
const keyword 34, 36, 57, 264, 277
block scoping 39
Constructor 90, 92
injection 454
contentHeight 176
contentWidth 176
Continuous integration (CI) 401
Control 252, 299, 381, 482
error handling 320
extend 354
IDs 268, 415
modify 154

Control flow 299
Controller 56, 103, 148, 252, 326, 346, 357
define 144
define class 168
fragment 166
inflation 148
lifecycle hook methods 144
mock methods 156
ProductWorklistController 239
Control locator 449
Coupled code 306
Create, read, update and delete (CRUD)
APIs 63
createContent 126–127
Cross-functional team 514–515
Custom control development 164
Custom error 315
Custom mapping 104
Custom rule 409–410
create 411
Cyclomatic complexity 417, 506
assess 421
metric 505

D

Dangling comma 348
Debugging 108, 388
Declaration 275
block 39
Declarative programming 313
Decorator 116
Deep copy 45
Default configuration 50
Default parameter 88
default-param-last rule 429
Default value 88
DefinitelyTyped 473
Definition file 355
definitions.d.ts 355
delete operator 35
Dependency injection 403, 441
Descriptive name 249, 262
Descriptor 120, 123, 128
attributes 128
Design pattern 252
Destructuring 291, 294, 405
Destructuring assignment 57
naming 58
Dialog 100, 174
close 177
declare in view 175

Dialog (Cont.)	
<i>design recommendations</i>	174
<i>display</i>	179
<i>implement in separate class</i>	176
<i>logic</i>	174
<i>open</i>	176
Do...while statement	303
Document Object Model (DOM) API	46, 409, 492
Dojo	510–511
Domain term	251
Don't repeat yourself (DRY) principle	433, 489
Driver	511, 513
Dummy	453
Duplication	433
Dynamic code check	399
Dynamic import	106
Dynamic typing	471
E	
ECMAScript	32
EditorConfig	365
Else if statement	300
Else statement	301, 310
Embedded expression	284, 286
Empty line	342, 367
Encapsulation	308
Enum	113, 358
<i>naming</i>	258
<i>types</i>	359
<i>TypeScript</i>	358
Enumerable own property	48
Enumeration	48, 113
<i>order</i>	48
Error cascading	72
Error class	318
Error handling	63, 315
<i>async/await</i>	82
<i>asynchronous code</i>	64
<i>best practices</i>	323
<i>controls</i>	320
<i>linters</i>	84
<i>messages</i>	318
<i>promises</i>	81, 84
Error message	55
Error object	317
Error swallowing	81
Escaping	43
ESLint	29, 34, 276, 285, 288, 379, 384, 392, 397, 399, 401–402, 409, 420–421, 426, 429, 433, 505
<i>additional mechanisms</i>	414
<i>install</i>	406
<i>no-return-await rule</i>	84
<i>return-await rule</i>	84
<i>rules</i>	402, 427
<i>TypeScript</i>	408
ES module	486
Event	125, 166, 259, 493
<i>parameter</i>	219, 264
EventBus	191
Event handler	82, 211, 259
<i>naming</i>	259
<i>wrapping</i>	83
Exception	81–82, 325
<i>unhandled</i>	316
Execution context parameter	212
Executor function	74
Exponent	283
Exponential notation	282
Exponential number	283
export keyword	105
Expression binding	346
extend method	92, 94, 97
F	
Faceless component	119
Factory function	104
False positive	435
Falsy value	89, 229, 289, 295, 300, 303, 311
Fan-in/fan-out	431
Feedback culture	507, 509
Fellowship	512
File placement	341
FIRST principles	444
Flag	309
Flaky tests	505
Floating point number	280
Folder structure	484
for...in statement	305
for...of statement	305
for await of statement	85, 88
Formatter	103, 151, 356, 363
<i>class</i>	258
<i>function</i>	151
<i>method</i>	346
Formatting	329, 347
<i>horizontal</i>	336
<i>indentation</i>	339

Formatting (Cont.)	
<i>style guide</i>	361
<i>tools</i>	363
<i>TypeScript</i>	351, 360
<i>vertical</i>	330
<i>XML views</i>	341
for statement	304
Four-eyes principle	398
Fragment	164, 341, 414
<i>control interface</i>	167
<i>custom controller</i>	165
<i>define dialog</i>	178
Freestyle applications	129
Fulfilled state	70, 72, 84
Fulfillment handler	70, 72
Function	26, 35, 40, 337, 350
<i>anonymous</i>	215, 247
<i>arity</i>	217
<i>arrow</i>	39
<i>bind to execution context</i>	212
<i>body</i>	237
<i>comparison</i>	264
<i>configureChart</i>	230
<i>confirm</i>	220–221
<i>constructors</i>	90
<i>declaration</i>	206
<i>default value parameter</i>	228–229
<i>define</i>	205
<i>destructuring assignments</i>	59
<i>do one thing</i>	238
<i>factory</i>	104
<i>formatName</i>	230
<i>formatting</i>	335, 340
<i>generator</i>	47
<i>getter and setter</i>	94, 99
<i>imperative style</i>	312
<i>invoke</i>	245
<i>iterable objects</i>	47
<i>jQuery.proxy</i>	213
<i>JSDoc</i>	379
<i>length</i>	430
<i>mimic import</i>	106
<i>naming</i>	255, 258
<i>native</i>	90
<i>nesting</i>	424
<i>object</i>	206, 208
<i>parameters</i>	49, 52, 217, 428
<i>reduce size</i>	242, 244
<i>SAPUI5 controller</i>	210
<i>scoping</i>	38
<i>sections</i>	385
<i>signature</i>	59, 89
Function (Cont.)	
<i>slicing</i>	419
<i>streamline parameters</i>	219–220
<i>sum</i>	225
<i>tag</i>	43
<i>typing</i>	111
Function.apply method	52
Fundamental Library	494
<i>Angular</i>	496
<i>current state</i>	498
<i>list component</i>	495
<i>React</i>	495
<i>stay updated</i>	498
G	
Generalists	514
Generator	47, 236
<i>asynchronous</i>	237
<i>range</i>	237
Generics	116, 476–477, 479
<i>error</i>	317
getResourceBundle() method	53, 76
Getter	94, 99–100, 213
Gilded rose kata	511
Git	25, 348, 390
Given-when-then	460
Globally unique identifiers (GUID)	282
Global object	91
Global scope	355
Global variable	267
Grades	502
Green grade	502
Grouping	332, 353
<i>break</i>	332
Guard clause	312
Guarding condition	310
H	
Habits	510, 513
Halstead Volume metric	436
Hierarchy	339
Hoisting	38, 275
Horizontal formatting	336
Hosting	107
HTML template	487
HTML views	143
Hungarian notation	270

I	
i18n.properties file	53, 55, 287
i18n method	169
if statement	300–301, 349
conditions	307
empty	306
nested	310
Immediately invoked function expression	
(IIFE)	246
Immutable object	36
Imperative programming	312
import keyword	105
Import statement	360
includes	31
Indentation	339
Index	262, 278
index.html	129
indexOf	31
Individual code ownership	500
Individual learning	503
init	126
Initialization	275, 296
Injection	453
Inline comment	377, 383
Inline logic	349
Instance annotation	138
Instance method	75, 208
Integer literal	280
big	282
Integration test	444
Interface	114, 351
define	188, 352
versus type	115
Internationalization	287
Interpolation	42, 284
I-shaped profile	514–515
Iterable object	46, 305
J	
JavaScript	27, 32, 90, 97, 105, 202–203, 224, 400
engine	62, 69, 90
file	179
function	205
linters	400
standard style	408
JavaScript Object Notation (JSON)	94, 128, 150, 458
model	100, 143
views	143
K	
Journey	252, 458
context	459
given-when-then	460
steps	459
jQuery	99
JSDoc	378, 383, 472
L	
Lazy singleton	372
Leadership styles	501
Learning techniques	510
Legacy code	27, 499
learning techniques	512
let keyword	39, 57, 277–278
Library	
descriptor	139
folder structure	140
projects	139
library.js	139
Line ending	348
Linter	84, 363, 400–401, 409
AST tree	412
automate	407
custom rules	410
disable	403
rules	402
XML	416
Linting	363, 399
folders	407
XML	414
List page	152
List report floorplan	134
Literal	60, 168, 275, 279
Log class	324
Logging	323
Loop	63, 303
asynchronous	88
do...while	303
for	304
for..in	305
for..of	305
indexes	278
naming	261
nested	261
while	303

M	
Magic number	279
avoid	36
Magic string	287
Maintainability	102, 330, 435
index	435
Managed property	215
Manager	509
manifest.js	287
Manifest.json	106, 120, 123, 129
Match array	60
Matcher	449
max-len rule	430
max-params rule	403, 406, 428
max-statements rule	430
Meaningful error message	469
Meaningful names	489
Merging	171
benefits	171
drawbacks	171
Message	319
box	51, 85, 322
custom	320
object	319
popover	323
strip	321
toast	320
view	322
messageBox function	50, 85, 110
advanced call	51
Message manager	319
Meta-comment	390
metadata.xml	458, 463
Metadata annotation	138
metadataLoaded method	41
metadata structure	93, 97, 99
Method	166, 259, 369
bind	212
byId	176
class	209
deleteProduct	213
destroy	191
exit	186
init	186
naming	258
navTo	223–224
onAfterClose	176
onBeforeOpen	176
onCopyProductPress	239
onExit	176
onInit	176
Method (Cont.)	
Promise.race	236
readUserDefaults	231
refactor	240, 242
sap.ui.getCore().attachLocalization	
Changed	191
sap.ui.getCore().attachThemeChanged	191
signatures	475
static	209
Metrics	505
Microsoft Edge	33
Microsoft Internet Explorer	33
Migration	475
Mixin	169–170
Mob programming	510, 513
Mocking	447
Mock object	447, 454
Model	56, 148, 252
creation	149
unavailable	149
Model-view-controller (MVC) pattern	142, 148, 252
Module	102, 147
cache	184
class	197, 200–201
comments	386
cyclomatic complexity	418
declare dependencies	190
dependencies	104
documentation	192
elements	103
execution	180
expose methods	186
fan-in/fan-out	431
formatting	334
handling settings	185
imports and exports	105
lifecycle	179
load	104, 182
mapping	103
modification	185
sap/m/MessageBox	194
service	197–199
single-page application	184
state	189
test	193
TypeScript	355
URL	104
Mozilla Developer Network (MDN)	33
Multiple languages	254
Mutation	35
testing	513

N

Namespace 249, 267, 356, 414, 481
TypeScript 355
 Naming 58, 249
alternative rules 271
casing 255
classes and enums 257
consistency 253
conventions 249, 288
functions and methods 258
prefixes 270
private members 265
shortening 254
strings 288
variables and parameters 260
 Navigator 511, 513
 Negative condition 307
 Nesting 78, 261, 310–311, 422
array 262
loop 261
rules 426
types 424
 new keyword 91
 Node.js 69, 399
 Node Package Manager (npm) 410
 Noise word 255
 no-magic-numbers rule 36
 Non-documented member 98
 Non-strict mode 281
 no-param-reassign rule 429
 Null 47, 295–296
 Number 35, 280
 Number of lines 430
 Number of statements 430
 Numeric separator 283–284

O

Object 35, 48, 293
class 96
clone 48
convert to list of values 44
error 317
global 91
manipulate 49
mutation 35
parameters 49
pass by reference 59
property 99
shorthand 293
state 36
thenable 77
 Object.assign API 48
 Object.defineProperty statement 99
 Object.freeze() 36
 Object.getOwnPropertySymbols method 98
 Object-oriented programming 90
 Object page 152
 Object page floorplan 134
 Object view 152
binding 153
 Observer pattern 149, 211
 Octal literal 281
 OData 137, 239, 257
grouping 67
message 320
 OData model 41, 75, 143, 425
V2 versus V4 63
 ODataModel class 63, 75
extend 75
 OData V2 mock server 457
test data 463
 One-Page Acceptance (OPA5) test 100, 147, 448–449, 461
action 449
assertion 449
naming 272
 OpenUI5 391, 415
 Options object 221–222
 Orange grade 502
 Overview page floorplan 136

P

package.json 268
 Page object 448, 451
 Pair programming 398, 510, 512
different roles 512
 Parallel calls 67, 72
 Parameter 49, 264, 275
comments 376
deconstructing 226–227
default 88
event 264
flags 309
mutation 442
naming 260
read-only 429
required 89
 parent property 94, 99
 Partitioning 164
 PascalCase 255, 257, 288
 Pending state 69
 Performance 69, 106, 399

Placeholder 53

multiple 55
 Polyfill 33, 107
 Positive condition 307
 Pragma comment 393
 prefer-rest-params rule 429
 Prefix 270, 414
code review 507
numbers 281
 Prettier 29, 360, 365, 367
 Private attribute 353
 Private member 94, 97
naming 265
special 98
 Programming by coincidence 387
 Promise 41, 51, 62, 69, 72, 85, 232, 422, 441
array 72
async/await syntax 234
chain 233
chaining 72, 78
create 74
detect 77
error handling 81
fulfilled 84
link 78
object 235
promisified code 234
promisify a method 75
racer 236
rejected 82
settled 74
states 73
 Promise.all method 63, 72, 85
 Property 166, 352
 Prototype 266, 456
cut off dependencies 456
inheritance 48, 90, 92, 373
pollution 75
 Public attribute 353
 Public method 125, 493
 Pure function 189

Q

QUnit 384, 448, 461, 467

R

React 487
 Readability 24–25, 31, 256, 340, 343, 399, 451
 readEntityAndRelated method 69
 read method 63
promisify 75

Redeclaration 276

Red grade 502
 Refactoring 28, 261, 349, 375, 386, 390, 424, 504
loops 28
 Reference 59
 RegExp constructor 290
 Regular expression 60, 289, 291
 Rejected state 70, 81
 reject function 74
 Rejection handler 71–72
 Required parameter 89
 resolve function 74
 ResourceBundle.getText API 53
 ResourceBundle object 54
 Resource model 143
 Rest parameter 225
versus argument 226
 Rest syntax 57, 61
 return keyword 40
 Return statement 312
 Reusability 187
 Reusable method 65
 Reverse tabnapping 46
 Root view 129
 Router 126

S

sap.m.MessageBox.show 49
 sap.m library 104
 sap.ui.base.Object base class 92, 94
 sap.ui.core.Component class 124
 sap.ui.define API 104
 sap.ui.require API 104–105
 SAP Business Application Studio 139
 SAP Cloud Application Programming
Model 473
 SAP Fiori 132, 450
floorplan 132, 134
 SAP Fiori elements 132
application generation 137
benefits 133
floorplan 136
framework 137
SAPUI5 applications 133
 SAP Fiori launchpad 126
 SAPUI5 23, 33, 92–93, 97–99
artifact 122
bootstrapping 129
common methods 259
component 119

SAPUI5 (Cont.)
data binding 142
error handling 318
library 147
mapping 103
reuse components 140
source code 94, 98
standalone application 180
versions 415
SAP Web IDE 139, 410, 415
Scenario test 444
Scoping 38, 78, 277
block 39
Selection list 496
Self-documenting code 369
Separation of concerns 440
Server-side message 319, 326
setError method 55, 77
simplify 61
Setter 94, 99, 101, 213
define 214
injection 454
startTime 214
Severity 401, 412
Shadow DOM 486
Shallow copy 45, 48, 292, 294
Shared vocabulary 250
Shift left principle 398, 439
Shorthand property 294
Signature 59, 104
default parameters 89
types 111
Single-responsibility principle 488
Sinon.JS 194, 455
test definition 455
siRequired flag 162
Slots 491
Smart control 457
SNAKE_CASE 256, 264
SOLID principles 114, 188, 201
Solution domain 252
Source code 94, 98, 103
Source mapping 108
Spacing 331, 337, 347
Splitting 424
Spread operator 50
Spread syntax 44, 56, 292, 294
arrays 44
function parameters 52
objects 48
Spy 454
Stable ID 268
Stack overflow 52
Standard style 408
Statement coverage 506
State message 319
Static application security testing (SAST) 399
Static class 257
Static code analysis 505
Static code check 397, 399
issues 504
Static file 179
Static method 97
extend 381
Stepdown rule 240
Strict mode 282
String 35
comparison 31
concatenation 42, 286
literal 43, 284, 287
Strong typing 109
Stub 453
Style guide 507–508
Subclass 92
define 92
inheritance 94
Subsequent request 64
super keyword 40, 93
switch/case statement 302, 420
Symbol 35, 98
Synchronous code 62
versions 69
System test 444

T

Tab 347
Tag function 43, 286–287, 392
Tagged template 286
Teams 501
learning 503
members 514
Technical backlog 406
Technical debt 504
Template 387
literal 42, 284, 286
Ternary operator 349
nesting 424
Testability 56, 60, 187, 440
Testable code 440
Test automate 440
FIRST principles 444
Test case 418, 459

Test data 461
choosing values 462
constants 462
naming 462
OData V2 mock server 463
Test-driven development (TDD) 387, 445–446
Testing 418, 421, 439
mutations 513
principles 440
Test isolation 440
Test method 458
names 458
Test pyramid 443
Thenable object 77
then method 70, 77–78
this keyword 40, 52, 93
throw statement 315
Transient message 319
Translation mechanism 53, 55
Transpiling 107–108, 111, 402
debugging 108
Truthy value 289, 300
try/catch block 81–82, 316
try statement 316
tsconfig.json 356
T-shaped profile 514–515
Type 110, 351, 355
check 107, 111
define 352
enum 113
guard 479–480
methods 475
versus interface 115
Typed views 143
typeof operator 207
TypeScript 90, 106, 174, 201, 351, 402, 407, 471–472
add to app 474
autocomplete 110
classes 353
classes in JavaScript modules 482
current state 472
ESLint 84, 408
example 475
features 110
formatting 351, 360
legacy features 485
module loading 480
refactoring 485
selection list 497
strong typing 109
transpiling and source mapping 108

U

UI5Object parameter 95
ui5-typescript-helloworld template 473
UI message 319
UIVeri5 448
Undefined 35, 47, 295–296
Union 352
Unit of measure 378
Unit test 443
User interface (UI)
component 119–120
controls 155
Utility 103
type 479
Utility class 176–177
ExampleDialog 177

V

Validation 55
message 320
Value type 35
Variable 38, 275
comments 376
declaration 275, 277
declare 38
formatting 333
global 39, 267
multiple 279
naming 57, 250, 260
reassignment 278
scoping 277
transfer 78
var keyword 38, 275–276
Vertical formatting 330
View 148, 176, 252, 341, 346, 414
ViewState class 100
Visual Studio Code 110, 360, 403, 413, 417
Vocabulary 138, 250

W

W3C 128
waitForEntities object 69

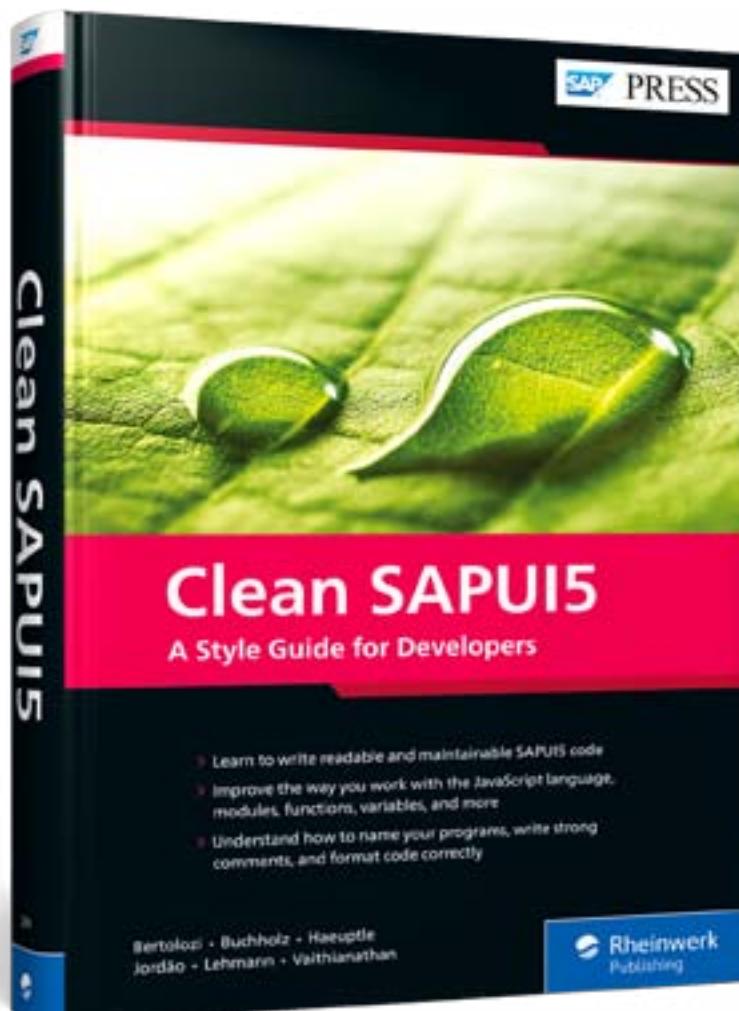
waitForOperation object 70, 72
Web Application Manifest 128
Web Audio API 371, 374
Web components 486
 APIs 490
 clean code 488
 current state 486
 example 489
 import 489
 properties and attributes 491
 slots 491
 staying updated 493
while statement 303
White box testing 194
White grade 502
Whitespace 332, 337
Worklist 136
Wrapping 65, 82, 85, 106

X

XML
 linting 414, 416
 model 143
 node 342
 schema 415
 schema definition (XSD) file 415
 tags 343
 views 143, 341
XMLComposite class 166

Y

Yellow grade 502
Yeoman 474



Daniel Bertolozzi, Arnaud Buchholz, Klaus Haeuptle, Rodrigo Jordão, Christian Lehmann, and Narendran Natarajan Vaithianathan

Clean SAPUI5: A Style Guide for Developers

530 pages, 2022, \$79.95

ISBN 978-1-4932-2228-5

 www.sap-press.com/5479



Daniel Bertolozzi is a software developer at SAP with more than 5 years of experience focused on SAP Fiori development. He has worked on several projects implementing SAPUI5 applications (in JavaScript and TypeScript) applying best practices for code quality.



Arnaud Buchholz is a development expert at SAP, with a focus on designing, developing, and enhancing SAPUI5 applications. He has more than 20 years of experience in software development.



Klaus Haeuptle is a lead architect, servant leader, coach, and community lead. During his career at SAP, he has worked as a developer and architect on several products based on various technologies.



Rodrigo Jordão is a development architect at SAP currently working on supply chain management and related solutions. He has spent his SAP career working on various SAP products, from industry-specific solutions like intellectual property management to foundational products like sales and distribution.



Christian Lehmann is a development architect working in the SAP S/4HANA quality area. During his 16 years at SAP, he has worked on the design, implementation, and test automation of SAP-based products, including applications based on SAPUI5.



Narendran NV is a senior developer, trainer, and security expert at SAP. Currently, he is part of a team that is focused on building advanced available-to-promise (ATP), which is an integral part of order fulfillment in SAP S/4HANA.

We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.